Exploring the LANDSCAPE of Distributed Graph Sketching

David Tench * Evan T. West † Kenny Zhang ‡ Michael A. Bender † Daniel DeLayo †
Martín Farach-Colton § Gilvir Gill † Tyler Seip ¶ Victor Zhang ‖

Abstract

Recent work has initiated the study of dense graph processing using graph sketching methods, which drastically reduce space costs by lossily compressing information about the input graph. In this paper, we explore the strange and surprising performance landscape of sketching algorithms. We highlight both their surprising advantages for processing dense graphs that were previously prohibitively expensive to study, as well as the current limitations of the technique. Most notably, we show how sketching can avoid bottlenecks that limit conventional graph processing methods.

Single-machine streaming graph processing systems are typically bottlenecked by CPU performance, and distributed graph processing systems are typically bottlenecked by network latency. We present LANDSCAPE, a distributed graph-stream processing system that uses linear sketching to distribute the CPU work of computing graph properties to distributed workers with no need for worker-to-worker communication. As a result, it overcomes the CPU and network bottlenecks that limit other systems. In fact, for the connected components problem, LANDSCAPE achieves a stream ingestion rate one-fourth that of maximum sustained RAM bandwidth, and is four times faster than random access RAM bandwidth. Additionally, we prove that for any sequence of graph updates and queries LANDSCAPE consumes at most a constant factor more network bandwidth than is required to receive the input stream. We show that this system can ingest up to 332 million stream updates per second on a graph with 2^{17} vertices. We show that it scales well with more distributed compute power: given a cluster of 40 distributed worker machines, it can ingest updates 35 times as fast as with 1 distributed worker machine. Graph sketching algorithms tend to incur high computational costs when answering queries; to address this LANDSCAPE uses heuristics to reduce its query latency by up to four orders of magnitude over the prior state of the art.

*Lawrence Berkeley National Laboratory
†Stony Brook University
‡Massachusetts Institute of Technology
§New York University
¶MongoDB
‖Meta Platforms

The full version of the paper can be accessed at https://arxiv.org/abs/2410.07518

Our code and experiments can be found at https://github.com/GraphStreamingProject/Landscape and https://doi.org/10.5281/zenodo.13845156

1 Introduction

Computing connected components is a fundamental graph-processing task with uses throughout computer science and engineering. It has applications in relational databases [80], scientific computing [62, 72], pattern recognition [31, 40], graph partitioning [49, 50], random walks [38], social network community detection [46], graph compression [39, 48], medical imaging [33], flow simulation [73], genomics [27, 57], identifying protein families [54, 78], microbiology [3], and object recognition [32]. Strictly harder problems such as edge/vertex connectivity, shortest paths, and k-cores often use it as a subroutine. Connected Components is also used as a heuristic for clustering problems [22, 23, 24, 61, 76, 77], pathfinding algorithms (such as Djikstra and A^*), and some minimum spanning tree algorithms. A survey by Sahu et al. [66] of database applications of graph algorithms reports that, for both practitioners and academic researchers, connected components was the most frequently performed computation from a list of 13 fundamental graph problems that includes shortest paths, triangle counting, and minimum spanning trees.

Computing the minimum cut of a graph (or equivalently its edge connectivity) is a closely related problem to connected components. It has applications in clusterings on similarity graphs [79, 30], community detection [12], graph drawing [41], network reliability [64, 43], and VLSI design [45].

The task of computing connected components or minimum cut becomes more difficult when graphs are ***dynamic***, meaning the edge set changes over time subject to a stream of edge insertions and deletions, and this task becomes harder still when the graphs are very large. Applications using dynamic graphs include identifying objects from a video feed rather than a static image [44] or tracking communities in social networks that change as users add or delete friends [9, 10]. Applications on large graphs include metagenome assembly tasks that may include gene

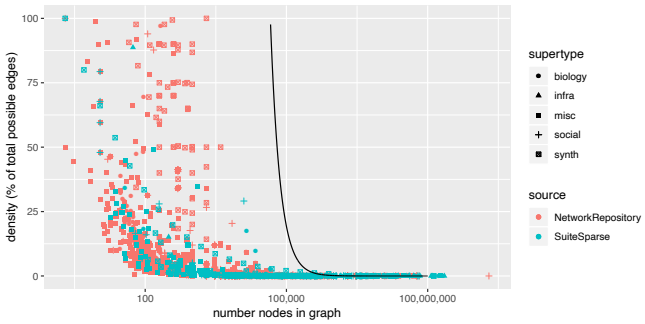


Figure 1: Graphs studied in academic works exhibit a selection effect. Any point to the left of the dark line indicates a dataset which can be represented as an adjacency list in 16GB of RAM.

databases with hundreds of millions of entries with complex relations [27], and large-scale clustering (a common machine learning challenge [24]). And of course graphs can be both large and dynamic. Indeed, Sahu et al.'s [66] database applications survey reports that a majority of industry respondents work with large graphs (> 1 million vertices or > 1 billion edges) and a majority work with dynamic graphs.

Dense-graph processing. The task of computing connected components is especially difficult for dynamic ***dense graphs***. The conventional wisdom is that massive graphs are always sparse, meaning that they have few edges per vertex. Tench et al. [71] contend that instead large, dense graphs do not appear in academic publications due to a selection effect: since we lack the tools to process these graphs, they are not studied. We expand on their survey of graph datasets to further support this claim. Figure 1 plots all graph datasets from the NetworkRepository [65] and SuiteSparse [17] collections. Note that nearly all the graphs can be stored as an adjacency lists using less than 16GB, and this pattern holds across repositories and across different types of graph dataset - including a variety of biological data, social networks, and infrastructure(e.g., road and computer networks). See Appendix F.1 for an expanded analysis which shows this selection effect even more strongly.

Despite this effect, there is evidence of dense graphs emerging in practical applications. Tench et al. note that "Facebook works with graphs with 40 million nodes and 360 billion edges. These graphs are processed at great cost on large high-performance clusters, and are consequently not released for general study" [71]. As another example, bipartite projection methods [56], commonly used in social sciences and bioinformatics, naturally generate large, dense graphs. Current techniques require storing these graphs in RAM, limiting the size of datasets analyzed this way — an explicit example of the selection effect [58]. The only way to conclusively determine whether there exist more applications that would benefit from dense-graph processing systems is to build such systems and see what applications emerge.

Tench et al. [71] demonstrate that ***linear sketching*** techniques [53, 1, 2, 29] can be used to process large, dense, and dynamic graphs. Linear sketching saves the most space when graphs are dense—this is because the size of a connectivity sketch of a V-vertex graph is $O(V \operatorname{polylog} V)$ and therefore is independent of the number of edges. This algorithm is representative of a large number of graph-sketching algorithms [53, 1, 2, 42, 29, 6] in the database theory literature that all share the same general structure.

However, the price for the small space of the sketches is high CPU cost: processing each update requires $O(\log^2 V)$ work, and the constants hidden by the asymptotic notation are large. Concretely, for a million-vertex graph, processing each edge ***update*** (an insertion or deletion) requires evaluating roughly 500 hash functions (in addition to other costs). Tench et al.'s implementation, GraphZeppelin, introduces some techniques for mitigating this high computational cost but ultimately their implementation is bottlenecked by CPU.

This paper. We design and implement distributed sketching algorithms for connected components and k-connectivity (or bounded k mincut) on dynamic graph streams. Using our implementation, we explore the surprising performance landscape of graph-sketching algorithms. Thus, we call this graph streaming system Landscape.

The peaks and valleys of this performance landscape will seem unfamiliar to graph-processing practitioners. The highest-order observation is that these techniques work well when graphs are dense and work poorly when they're sparse. This is in contrast to most techniques which work better when graphs are sparse. We also show these sketching algorithms can avoid computational bottlenecks that are unavoidable using conventional graph-processing techniques, but these algorithms also struggle in some cases where traditional algorithms excel.

We summarize these findings in Claim 1 in Section 2, but for context we first briefly survey the bottlenecks in graph-stream processing and how they affect existing graph-processing systems (which are designed for sparse graphs).

1.1 Three Bottlenecks in Graph-Stream Processing

Graph-stream processing systems can be bottlenecked on any of three resources: ***space***, ***CPU***, and ***communication***. Even a single-worker system requires some network communication; at minimum it must receive the input stream from a network link.

Some systems, such as Aspen [18] and Terrace [60], are bottlenecked on ***space***. These systems maintain lossless representations of the input graph in RAM on a single machine, and such lossless representations can be large. These systems are optimized for processing sparse graphs

Table 1: Summary of ingestion bottlenecks. Existing graph systems are bottlenecked by space, CPU, or network communication costs when processing dense graph streams. In contrast, in this paper we present a system that overcomes these three bottlenecks.

	space	CPU	Network
single-machine lossless [18, 60]	⋆ $\Theta(V^2)$	⋆ ⋆	N/A
single-machine sketching [71]	⋆ ⋆ ⋆ $\Theta(V \log^3(V))$	⋆	N/A
distributed lossless [13, 34, 74, 35, 36, 26]	⋆ ⋆ $\Theta(V^2)$	⋆ ⋆	⋆
distributed sketching (this paper)	⋆ ⋆ ⋆ $\Theta(V \log^3(V))$	⋆ ⋆	⋆ ⋆ ⋆

⋆ = bottleneck; ⋆ ⋆ = good; ⋆ ⋆ ⋆ = optimal

where this memory burden is less onerous but struggle when processing large, dense graphs.

Other systems, such as GraphZeppelin [71], are bottlenecked on ***computation*** (as we explain earlier). GraphZeppelin maintains a lossily-compressed graph representation, reducing storage requirements (and improving performance on dense graphs) at the cost of higher CPU load for processing updates and answering queries.

Distributed graph-stream processing systems are bottlenecked by network ***communication***. For dynamic systems [74], the input stream is received at a central node called the ***main node*** and information about the graph is distributed across the cluster to ***worker nodes***. For non-dynamic systems the data is distributed among worker nodes before computation begins. Since the graph data is spread across many worker nodes, and graphs often have poor data locality, most computation on graphs require worker nodes to send lots of information to each other [13, 34, 74, 35, 36, 26]. This is an example of a general challenge in distributed database systems. These systems scale by spreading data among the aggregated memory of nodes in a cluster at the cost of high inter-node communication, which can increase query and transaction latencies [81, 19, 20].

Table 1 summarizes the bottlenecks for each of these approaches.

Objective standard for update performance: RAM bandwidth. One issue with evaluating the performance of a distributed system to compute fully-dynamic (i.e., edges can be inserted or deleted) graph connectivity is that there are no other open-source distributed systems that solve this precise problem (see Appendix G). However, for stream ingestion rate we can compare against objective upper bound.

The update performance of any graph-streaming system is limited by the ***data acquisition cost***, that is, the cost for the main node to simply read the entire input stream. Even if we ignore the cost to update the connectivity information in the graph data structures, we still need to read the input. Thus, an objective standard describing the ideal performance is simply the RAM bandwidth. In fact, there are two notions of RAM bandwidth: ***random-access RAM bandwidth*** (the speed at which we can write words to random locations) and ***sequential-access RAM bandwidth*** (the speed at which we can write words to sequential locations).

Since graphs have notoriously poor data locality, an update rate close to random-access RAM bandwidth is a natural goal for a graph stream-processing system. Sequential-access RAM bandwidth is truly a bound on the best possible update rate because any stream-processing system must write the input data into memory, that is, it must pay the data acquisition cost.

2 Results

In this paper, we build the Landscape graph-processing system, which is optimized for dense, dynamic graphs. We use this system to establish how graph sketching can overcome classical graph processing bottlenecks.

2.1 The Landscape Graph-Processing System

We build Landscape, a linear-sketch-based distributed graph-stream processing system that computes connected components and k-connectivity of dynamic graphs.

Landscape keeps its sketch (a lossily compressed graph representation) on the main node and distributes the CPU work of processing updates.

Because these linear sketches are small (size $\Theta(V \log^3 V)$), they fit on a single node, even when the graph is dense. The work to maintain these sketches can be chunked off into large batches that can be computed independently by worker nodes. As a result, Landscape avoids the CPU bottleneck because it has lots of worker nodes to help, and Landscape avoids the communication bottleneck because the communication cost is amortized away by the CPU cost to process a batch.

In fact, we prove theoretically that for any sequence of graph updates and queries Landscape's total communication cost is only a small, user-configurable constant (by default, four) factor larger than the cost for the main node to receive the input stream. Additionally, we introduce a new sketching algorithm, CameoSketch, which requires only $O(\log V)$ distributed work per update (compared to $O(\log^2 V)$ for the prior state of the art), which allows the algorithm to scale more rapidly with limited cluster resources.

Performance. Landscape achieves the following:

- Landscape is able to process graph streams only 4.5× slower than the multi-threaded RAM sequential-write bandwidth, the objective upper bound on insertion

performance for any system that receives the input stream at the main node. This is more than four times faster than random access RAM bandwidth.
- We show that LANDSCAPE can ingest up to 332 million stream updates per second on a graph with 2^{17} vertices.
- We show that it scales well with more distributed compute power: given a cluster of 40 distributed worker machines, it can ingest updates 35 times as fast as with 1 distributed worker machine.
- We experimentally verify that LANDSCAPE uses at most $4\times$ the network bandwidth required to read the input stream.
- LANDSCAPE's GREEDYCC query heuristic reuses partial information from prior query results, achieving up to a four orders-of-magnitude reduction in query latency.

Outperforming lossless representations on dense graphs. To put this performance in context, consider a simpler task: maintaining an adjacency matrix of the graph defined by the input stream.[1] If the graph is dense and edges are random, an adjacency matrix is essentially the space-optimal lossless graph representation. We ignore the cost of answering queries, which an adjacency matrix does not efficiently support.

LANDSCAPE's graph-sketch representation is smaller than this adjacency matrix even when the input graph has only 310,000 vertices. Even more interestingly, LANDSCAPE's update throughput is also faster than the update throughput of the adjacency-matrix representation—which is just a single bit flip per edge. We emphasize: one of the most dramatic advantages of distributed graph sketching is that updates are faster than adjacency-matrix updates even when the entire adjacency matrix fits in RAM.

LANDSCAPE's updates are fast not because they are small (you cannot beat a single bit flip), but because the CAMEOSKETCHES have good data locality—and the edge updates result in primarily sequential accesses to RAM on the main node, rather than random access. Said differently, LANDSCAPE processes more edge updates per second than it is possible to flip bits in random locations in RAM.

2.2 Circumventing the Classical Bottlenecks for Graph-Stream Processing

The performance implications of sketching for dense graph processing are encapsulated in the following claim. Throughout this paper, as we present theoretical and experimental results, we will refer to the element of the claim they support.

[1] Note that while adjacency matrices may be compact, they have high query latency. Even disregarding this limitation, we see that adjacency matrices are outperformed by sketching on dense graphs.

CLAIM 1. (**Dense graph processing**)

1. ***Space consumption.*** *Graph sketches for dynamic, massive, dense graphs can be maintained so that they use less space than traditional graph-storage methods. E.g.,* LANDSCAPE *is asymptotically space-optimal and its new* CAMEOSKETCH *algorithm uses 29% of the space of the prior state-of-the-art [71].*

2. ***CPU cost.*** *Graph sketches have high CPU cost to update, but this cost can be distributed away. E.g.,* LANDSCAPE*'s* CAMEOSKETCH *algorithm reduces the asymptotic work per update from $O(\log^2 V)$ (the prior SOTA) to $O(\log V)$. We show that this yields a $7\times$ increase to update throughput in experiments.*

3. ***Communication costs.*** *The above distribution of CPU work can be done with nearly optimal communication: specifically, the total communication cost is only a small constant number of times larger than the data acquisition cost. We prove this theoretically and validate it experimentally.*

4. ***Stream-ingestion can be blindingly fast–nearly the universal speed limit.*** *There is a universal speed limit for stream ingestion, which is simply the cost to write the stream sequentially into RAM. A graph-sketch based system for connectivity and k-connectivity can match this bound within a remarkably small constant factor.* LANDSCAPE *ingests graph data at a rate that is within a factor 4 of sequential RAM bandwidth.*

Table 1 summarizes the strengths of the sketching approach: LANDSCAPE ingestion is not bottlenecked on space, CPU, or communication.

The following observations are nearly immediate and unsurprising, and we include them for completeness.

Queries. Similar to the trade-off between reading and writing in (nongraph) databases [59, 8, 67, 7], ingesting a graph stream and processing it into a sketch is faster than querying the sketch. LANDSCAPE answers each query in single digit seconds, even when there are 37 billion edges and 2^{19} vertices, using a combination of provable worst-case query algorithms, accelerated with powerful heuristics.

Sparse graph-processing via sketching. Graph sketching is not the best solution for graphs that are relatively sparse. On sparse graphs, the LANDSCAPE approach retains its asymptotic guarantee of low communication but is not space-efficient and may not be able to distribute away its CPU costs. For completeness we evaluate LANDSCAPE's performance on sparse graphs and validate these theoretical predictions.

3 Preliminaries & Definitions

The Graph Streaming Model. In the ***graph semi-streaming*** model [25] (sometimes just called the ***graph streaming*** model), an algorithm is presented with a ***stream*** S of updates (each an edge insertion or deletion) where the length of the stream is N. Stream S defines an input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $V = |\mathcal{V}|$ and $E = |\mathcal{E}|$. The challenge in this model is to compute (perhaps approximately) some property of $\mathcal{G}$ given a single pass over S and at most $O(V \operatorname{polylog} V)$ words of memory. Each update has the form $((u, v), \Delta)$ where $u, v \in \mathcal{E}, u \neq v$ and $\Delta \in \{-1, 1\}$ where 1 indicates an edge insertion and -1 indicates an edge deletion. Let s_i denote the ith element of S, and let S_i denote the first i elements of S. Let $\mathcal{E}_i$ be the edge set defined by S_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 $e \in \mathcal{E}_{i-1}$.

In this paper, we consider the following two problems in this model:

PROBLEM 1. (**Streaming Connected Components.**) *Given an insert/delete edge stream of length N that defines a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, return a spanning forest of $\mathcal{G}$.*

PROBLEM 2. (**Streaming k–Edge Connectivity.**) *Given an insert/delete edge stream of length N that defines a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, for any cut $C \subset \mathcal{V}$, return the cardinality of cut C (denoted by $w(C)$) if $w(C) < k$, and return ∞ otherwise.*

Prior work [1, 71] has also considered these problems in the graph streaming model.

A note on the query model for Landscape. The graph streaming model above requires computing an answer at the end of the stream. In contrast, LANDSCAPE can answer connectivity queries interspersed arbitrarily among insertions and deletions during the stream. It answers both ***global connectivity*** queries, where the task is to return the connected components or k-edge connectivity of the graph, and ***batched reachability*** queries, where the query consists of a set of vertex pairs $(u_1, v_1), (u_2, v_2)...(u_k, v_k)$ and the task is to determine whether u_i is in the same connected component as v_i for each $i \in [k]$. The goal is to minimize query latency in addition to the goals of minimizing space use and maximizing stream ingestion considered by prior work. For more information see Section 5.3. We assume that the stream is not adaptive, meaning that edge updates and connectivity queries do not depend on the results of prior queries.

Other models in streaming connected components systems. Aspen [18] and Terrace [60] are graph-stream processing systems which support connectivity queries. Unlike the semi-streaming model proposed by [1] above, these systems work in the ***batch-dynamic model***, where updates are applied to a non-empty graph in batches exclusively containing insertions or deletions. The minimum size of batches is a parameter which affects system performance. Connectivity queries may be issued in between batches, but not during a batch. Aspen is capable of computing queries concurrently with processing updates while Terrace is not. In contrast, LANDSCAPE is designed to handle arbitrarily interspersed updates and queries with no notion of batched input. Like Terrace, it does not compute queries concurrently with processing updates.

4 Sketching Graphs

In this section we briefly review the prior work on connectivity sketching, and then present CAMEOSKETCH, an improved sketching subroutine that reduces the update cost from $O(\log^2 V)$ to $O(\log V)$, and reduces the sketch size by a significant constant factor (73%).

4.1 Prior Work

Ahn et al. [1] initiate the field of graph sketching with their connected components sketch, which solves the streaming connected components problem in $O(V \log^3 V)$ space. A key subproblem in their algorithm is ***ℓ_0-sampling***: a vector x of length n is defined by an input stream of updates of the form (i, Δ) where value Δ is added to x_i, and the task is to sample a nonzero element of x using $o(n)$ space. They use an ℓ_0-sampler (also called an ℓ_0-sketch) due to Cormode et al.:

THEOREM 4.1. *(Adapted from [16], Theorem 1): Given a 2-wise independent hash family $\mathcal{F}$ and an input vector $x \in \mathbb{Z}^n$, there is an ℓ_0-sampler using $O(\log^2(n) \log(1/\delta))$ space that succeeds with probability at least $1 - \delta$.*

We denote the ℓ_0 sketch of a vector x as $\mathcal{S}(x)$. The sketch is a linear function, i.e., $\mathcal{S}(x) + \mathcal{S}(y) = \mathcal{S}(x + y)$ for any vectors x and y. Ahn et al. define a ***characteristic vector*** $f_u \in \mathbb{Z}_2^{\binom{V}{2}}$ of each vertex $u \in \mathcal{V}$ such that each nonzero element of f_u denotes an edge incident to u. That is, $f_u \in \mathbb{Z}_2^{\binom{V}{2}}$ s.t. for all vertices $0 \leq i < j < V$:

$$f_u[(i,j)] = \left\{ \begin{array}{ll} 1 & u \in \{i, j\} \text{ and } (i,j) \in \mathcal{E} \\ 0 & \text{otherwise} \end{array} \right\}$$

Further, for any $S \subset \mathcal{V}$, the nonzero elements of $f_S = \sum_{u \in S} f_u$ are precisely the set of edges crossing the cut $S, \mathcal{V} \backslash S$. For each $u \in \mathcal{V}$, the algorithm computes $\mathcal{S}(f_u)$: for each edge update (u, v, Δ) it computes $\mathcal{S}(e_{(u,v)})$ where $e_{(u,v)}$ denotes the vector with 1 in position idx $= (u, v)$ and 0 in all other positions. It maintains $\mathcal{S}(f_u) = \sum_i \mathcal{S}(x_i)$ for each u. Then for arbitrary $X \subset \mathcal{V}$ they can sample an edge across the cut $X, \mathcal{V} \setminus X$ by computing $\mathcal{S}(f_X) = \sum_{u \in X} \mathcal{S}(f_u)$.

This allows them to perform Borůvka's algorithm using the sketches: they form $O(\log V)$ ℓ_0-sketches

$\mathcal{S}_0(f_u), \mathcal{S}_1(f_u), \cdots, \mathcal{S}_{O(\log V)}(f_u)$ for each u. We call $\mathcal{S}(f_u) = \bigcup_{i\in[O(\log V)]} \mathcal{S}_i(f_u)$ the ***vertex sketch*** of u, and it has size $O(\log^3 V)$. Then $\forall u \in \mathcal{V}$ they query $\mathcal{S}_1(f_u)$ to sample an edge incident to u. For each resulting component X they compute $\mathcal{S}_2(f_X)$ and repeat until all connected components are found. Since each vertex sketch $\mathcal{S}(f_u)$ has size $O(\log^3 V)$ bits, the entire data structure has size $O(V \log^3 V)$. See Appendix A for a more complete description of this algorithm, along with an illustrative example.

Testing k-connectivity. Ahn et al. [1] also show how to test k-connectivity of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ (that is, to exactly compute the minimum cut of $\mathcal{G}$ provided this value is $\leq k$) by constructing a k-connectivity certificate $H = \bigcup_{i\in[k]} F_i$ where $F_0, F_1, \ldots, F_{k-1}$ are edge-disjoint spanning forests of $\mathcal{G}$. H has the property that it is k'-edge connected iff G is k'-edge connected for all $k' \leq k$. They find each F_i by computing k connectivity sketches of $\mathcal{G}$ in a single pass over the stream. After the stream, the first connectivity sketch is used to find F_0 and the edges of F_0 are deleted from the remaining $k-1$ connectivity sketches. The second connectivity sketch can now be used to get F_1, whose edges are subsequently deleted from the remaining $k-2$ sketches and so on. The size of the sketches is $O(kV \log^3 V)$ bits.

GraphZeppelin. Tench et al. [71] present GraphZeppelin, the first implementation of Ahn et al.'s connected components algorithm which uses a faster ℓ_0-sketch algorithm which they call CubeSketch. GraphZeppelin also uses an external-memory-optimized data structure called a ***gutter tree*** to I/O-efficiently collect updates to be processed, allowing the algorithm to run quickly even when the sketches are stored on disk.

To achieve a failure probability of δ, GraphZeppelin's CubeSketch uses $O(\log^2(n)\log(1/\delta)$ bits of space and has worst-case update time $O(\log(n)\log(1/\delta))$. As in Ahn et al., they use $O(\log V)$ of these sketches for each vertex and set δ to be a small constant. So the space cost per vertex is $O(\log^3 V)$ bits and the worst-case update time is $O(\log^2 V)$. See Appendix B for a detailed description of CubeSketch.

4.2 Landscape's new sketch: CameoSketch

We develop a new ℓ_0 sampler called CameoSketch for use in Landscape. CameoSketch improves upon CubeSketch with a new update procedure that is a $O(\log V)$ factor faster to update and reduces space usage by a constant factor via a refined analysis. All other details, including the query procedure, remain unchanged. We present the full details of CameoSketch in Appendix B and report the asymptotic performance here.

Update procedure. CameoSketch uses a simpler and faster update procedure than CubeSketch.

Theorem 4.2. CameoSketch *is an ℓ_0-sampler that, for vector $x \in \mathbb{Z}_2^n$, uses $O(\log^2(n)\log(1/\delta))$ space, has worst-case update time $O(\log(1/\delta))$, and succeeds with probability $1-\delta$.*

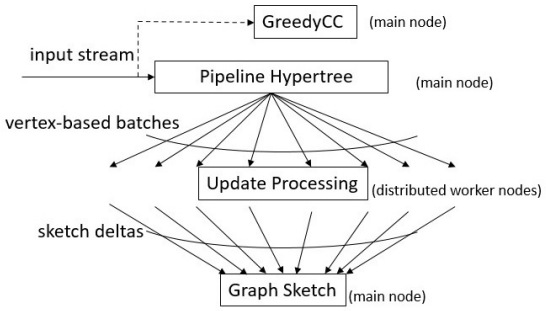


Figure 2: Data flow diagram for Landscape's ingestion algorithm. Connectivity information from the input stream is compressed into the graph sketch and added to GreedyCC.

Theorem 4.2 demonstrates that CameoSketch reduces the CPU burden of performing updates and supports Claim 1.2. The proof of this theorem can be found in Appendix B.

Reduced constant factors. In GraphZeppelin, Tench et al. use $56\log(1/\delta)\log n$ bytes of space to guarantee a failure probability of at most δ when sketching a vector of length $n < 2^{64}$ using CubeSketch. Via a careful constant-factor analysis, we can show that CameoSketch can match this failure probability with significantly less space:

Theorem 4.3. *Using 3-wise independent hash functions, CameoSketch requires $8\log_3(1/\delta)(\log n + 5)$ bytes of space to return a nonzero element of a length $n < 2^{64}$ input vector w/p at least $1-\delta$.*

Theorem 4.3 immediately implies a space savings of up to 90% compared to CubeSketch and thus supports Claim 1.1. In our implementation, we conservatively choose to use slightly more space than this theorem requires to reduce the failure probability further. Still, our implementation requires only 2/7ths of the space used in GraphZeppelin [71] (see Section 6 for details).

See Appendix H for the proof of this theorem.

5 Landscape Design

Landscape uses CameoSketches to compute connectivity. The CPU work of computing updates to $\mathcal{S}(\mathcal{G})$ is done by distributed workers, while $\mathcal{S}(\mathcal{G})$ itself is stored on the main node. To answer queries, the main node computes a spanning forest via Borůvka's algorithm using $\mathcal{S}(\mathcal{G})$ as described in Section 4 or via a heuristic algorithm which we call GreedyCC and describe in Appendix E.4.

In this section we describe how LANDSCAPE's main node efficiently collects updates into vertex-based batches to minimize communication, how the distributed workers process these batches of updates into sketch form, and how LANDSCAPE answers connectivity queries. We also prove the asymptotic upper bounds on the CPU and communication costs incurred by these operations. This analysis explains LANDSCAPE's surprising performance profile.

Figure 2 summarizes the LANDSCAPE data flow. The input stream, consisting of edge insertions, edge deletions, and queries arrives at the main node. Updates (insertions and deletions) are inserted into the ***pipeline hypertree*** (Section 5.1.2) and also into GREEDYCC (Section E.4). The pipeline hypertree collects updates into ***vertex-based batches*** (Section 5.1.1) which are sent to distributed worker nodes. These worker nodes process batches by running the CAMEOSKETCH algorithm, producing ***sketch deltas*** (Section 5.2) which are applied to the CAMEOSKETCHES on the main node.

5.1 Ingesting Stream Updates on the Main Node

5.1.1 Vertex-Based Batching The core technique that makes distributed sketch processing communication-efficient (and therefore feasible) is *vertex-based batching*, where many updates with a common endpoint are collected into a batch. Intuitively, because one or many updates to the same endpoint can be represented as a single sketch delta of fixed size, batching updates by endpoint drastically reduces the communication cost of sending sketch deltas from worker nodes to the main node.

Specifically, any updates for $u \in \mathcal{V}$ are collected into a batch $B_u \subseteq \{(x, y) \in \mathcal{E} \mid x = u \vee y = u\}$. B_u is sent to a single distributed worker, which returns a sketch of the updates. As we will see in Section 5.2 this sketch has size $\phi = O(\log^3 V)$ bits. During update processing, LANDSCAPE only sends B_u when $|B_u| \geq \alpha\phi/\log V$ for some constant $\alpha \geq 1$. Each update requires $\log V$ bits to represent, so a buffer that contains $\alpha\phi/\log V$ updates has size $\alpha\phi$ bits.

As a result of this policy, the amortized communication cost per update is small. Say the stream contains N updates. The bandwidth cost to receive the input stream is N. Since each update is included in two vertex-based batches (one per endpoint) and each batch is sent to a distributed worker once, the total bandwidth cost of sending vertex-based batches is $2N$. Finally, each vertex-based batch induces the distributed worker that receives it to respond with a sketch delta (which is $1/\alpha$ of the batch's size). This means that as long as LANDSCAPE is processing full vertex-based batches, the network bandwidth cost of processing N updates is at most $(3 + 1/\alpha)N$. This technique is simple, but crucial for good performance.

5.1.2 Pipeline Hypertree To make vertex-based batching fast, we design the ***pipeline hypertree***, which is a simplified and parallel variant of the buffer tree [5] designed to minimize cache line misses and thread contention. The pipeline hypertree receives arbitrarily ordered stream updates and consolidates them into vertex-based batches. Each update inserted into the pipeline hypertree is moved $O(\log_{C/\mathcal{L}} V)$ times before being returned in a vertex-based batch, where C denotes the size of L3 cache and $\mathcal{L}$ denotes the size of an L3 cache line. The total size of the data structure is $O(V \log^3 V)$ bits. We defer description of the design and implementation of the pipeline hypertree to Appendix C.

5.2 Distributed Sketch Processing

CAMEOSKETCHES have strong data locality, which we exploit for parallelism: processing a graph edge update (u, v, Δ) requires updating only $\mathcal{S}(f_u)$ and $\mathcal{S}(f_v)$ and these updates can be performed independently of each other. Because the sketches are linear, the sketch update for (u, v) can be computed on its own and later summed to the sketch of u. This means that the vast majority of the computation required to process (u, v) can be performed before accessing the sketches for u and v. LANDSCAPE exploits this independence to distribute the computational cost of these updates while storing the sketches on a single worker.

When a batch of updates $(e_1, e_2, ...)$ for vertex u is sent to a worker node, the worker node computes $\sum_j \mathcal{S}(e_j)$, which we call a ***sketch delta*** (since it is a sketch encoding the change in the neighborhood of vertex u. Note that it has size $O(\log^3 V)$—equal to the size of a vertex sketch. This immediately gives the following result:

THEOREM 5.1. (DISTRIBUTED COST.) *The distributed CPU cost of processing a batch of x updates for vertex $u \in \mathcal{V}$ into a sketch delta $\mathcal{S}_u$ is $O(x \log(V))$.*

Sketch merging. LANDSCAPE's main node maintains the ***graph sketch***: $\mathcal{S}(\mathcal{G}) = \bigcup_{u \in \mathcal{V}} \mathcal{S}(f_u)$. After a sketch delta for vertex u is created by a distributed worker is then sent to the main node where it is added to $\mathcal{S}(f_u)$. The graph sketch is stored in RAM which is feasible since it has total size $O(V \log^3(V))$ bits.

5.3 Finding Connected Components

LANDSCAPE is designed to efficiently answer two types of connected component queries: ***global connectivity queries*** where the task is to map each vertex to its connected component, and ***batched reachability queries*** where the query consists of a set of vertex pairs $(u_1, v_1), (u_2, v_2)...(u_k, v_k)$ and the task is to determine whether u_i is in the same connected component as v_i for each $i \in [k]$. At a high level, LANDSCAPE answers these queries by producing a spanning forest of the

graph defined by the input stream. This is done as described in Section 4 via Boruvka's algorithm on the vertex sketches.

Processing queries can increase network bandwidth costs if not handled carefully. Computing the spanning forest from the sketches can only be done after all pending stream updates have been processed. We say the graph sketch stored on the main node is ***current*** with respect to query q at time t if there are no pending updates; i.e., all updates that arrived prior to time t have been processed and merged into the graph sketch. If for some vertex i there are a small number of updates for i in the pipeline hypertree, then by the reasoning in Section 5.1.1 this could incur an average communication cost per update of $O(\log^2 V)$. In the worst case, the input stream could insert a perfect matching into an empty graph and then issue a query; the average communication cost per update would be $O(\log^2 V)$.

GLANDSCAPE avoids this problem by adopting a hybrid distribution policy for pending updates when a query is issued. When a query is issued, GLANDSCAPE first flushes the pipeline hypertree so all pending updates are stored in the leaves. For each leaf, if the leaf is at least a γ-fraction full, it is sent as a vertex-based batch to a distributed worker, where $\gamma \in (0, \frac{1}{2}]$ is a parameter chosen by the user. All leaves that are less than a γ-fraction full are processed locally on the main node, costing no additional network bandwidth.

As a consequence of this policy, we have the following theorems.

THEOREM 5.2. (COMMUNICATION COST.) *The communication cost of ingesting N updates and answering Q queries is at most $(3 + 1/(\gamma\alpha))N$, where γ and α are constants.*

Proof. See Appendix D. ∎

Importantly, this means that GLANDSCAPE never uses more than a constant multiple of the network bandwidth required to receive the input stream, *regardless of the number or distribution of queries.*

THEOREM 5.3. (COMPUTATIONAL COST ON MAIN NODE.) *Given a input stream of length N defining $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a series of Q connectivity queries issued throughout the stream, let N_i denote the number of edge updates that arrive after query Q_i but before query Q_{i+1}.* GLANDSCAPE *never uses more than $O(V \log^3 V)$ bits of space on the main node while processing the stream, and the amortized cost per update to process N_i is $O(\log_{C/\mathcal{L}}(V))$ if $N_i = \Omega(V \log^2(V))$ and $O(\log(V))$ otherwise. Computing each query Q_i takes $O(V \log^2(V))$ time.*

Proof. See Appendix D. ∎

As a result, when $N_i = \Omega(V \log^2 V)$ the amortized CPU cost for all computation on the main node is $O(\log_{C/\mathcal{L}}(V))$ (the amortized cost to process updates with the pipeline hypertree). For reasonable values of C and $\mathcal{L}$ this logarithm evaluates to a small constant, typically 3. Moreover, these operations are just data movement operations, so they can be done at near RAM bandwidth.

5.4 Computing k-connectivity GLANDSCAPE's architecture can in principle accomodate many other graph sketch algorithms that use connectivity as a subroutine. For instance, since the k-connectivity sketch requires maintaining k independent copies of the connectivity sketch, we can achieve comparable computation and communication upper bounds by slightly modifying the procedure used to distribute connectivity sketching. Set the size of a vertex-based batch and the size of leaf node buffers in the pipeline hypertree to $\alpha \cdot k \log^3 V$ (the per-vertex sketch size for k-connectivity). When a distributed worker recieves a vertex-based batch, it computes the sketch delta of the batch for all k copies of the connectivity sketch and sends this back to the main node. To answer k-connectivity queries, GLANDSCAPE produces a k-connectivity certificate using the query algorithm summarized in Section 4. This immediately gives the following result:

THEOREM 5.4. (K-CONNECTIVITY.) *Given a input stream of length N defining $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a series of Q connectivity queries issued throughout the stream, let N_i denote the number of edge updates that arrive after query Q_i but before query Q_{i+1}.* GLANDSCAPE *never uses more than $O(kV \log^3 V)$ bits of space on the main node while processing the stream. The main node amortized cost per update to process N_i is $O(\log_{C/\mathcal{L}}(V))$ if $N_i = \Omega(kV \log^2 V)$ and $O(k \log V)$ otherwise. Computing each query Q_i takes $O(k^2 V \log^2 V)$ time. The distributed CPU cost of processing a batch of x updates for vertex $u \in \mathcal{V}$ into a sketch delta $\mathcal{S}_u$ is $O(xk \log V)$. The communication cost of ingesting N updates and answering Q queries is at most $(3 + 1/(\gamma\alpha))N$, where γ and α are constants.*

Note that the network communication cost does not increase above that of connectivity, and for sufficiently infrequent queries the cost to the main node is also independent of k. See Section 7.4 for experimental confirmation of these results.

6 Landscape Implementation

Processing stream updates into the graph sketch is a computationally intensive process: for a moderately sized data-set with 2^{18} vertices, applying a single edge update requires evaluating 184 hash functions. GLANDSCAPE farms out this computationally intensive portion of the workload to worker nodes while the other portions of stream ingestion, including update buffering and sketch storage, remain the responsibility of the main node. GLANDSCAPE uses $164V * (\log^2 V - \log V)$B of space on the main node

to store the sketches and the pipeline hypertree. On the worker nodes, LANDSCAPE requires storage for a single sketch and batch per CPU. Thus, a worker node with t threads requires $t \cdot 164(\log^2 V - \log V)$ bytes. On a billion-vertex graph, each worker thread requires only 64 KiB of RAM. When computing k-connectivity, all of the above costs are multiplied by a factor k.

LANDSCAPE must handle two tasks: stream ingestion, where edge updates from the input stream are compressed into the graph sketch; and query processing, where connectivity queries are computed from the graph sketch (or sometimes from auxiliary query-accelerating data structures, described below). We defer a full description of the implementation, including auxiliary data structures, parameter choices, and software tools, to Appendix E.

Table 2: Datasets used in our experiments.

Name	Vertices	Edges	Stream Updates
kron13	2^{13}	1.7×10^7	1.2×10^8
kron15	2^{15}	2.7×10^8	1.9×10^9
kron16	2^{16}	1.1×10^9	7.7×10^9
kron17	2^{17}	4.3×10^9	3.1×10^{10}
ca-citeseer	2.3×10^6	8.1×10^5	1.1×10^8
p2p-gnutella	6.3×10^4	1.5×10^5	1.9×10^6
rec-amazon	9.2×10^4	1.3×10^5	1.7×10^6
google-plus	1.1×10^5	1.4×10^7	1.9×10^8
web-uk-2005	1.3×10^6	1.2×10^8	1.6×10^9
erdos18	2^{18}	1.7×10^{10}	4×10^{10}
erdos19	2^{19}	3.4×10^{10}	4×10^{10}
erdos20	2^{20}	8×10^{10}	1×10^{11}

7 Experiments

Experiment setup. We implemented LANDSCAPE in C++14 and compiled using g++ version 9.3 with openmpi 4.1.3 for Linux. We ran our experiments on an AWS cluster composed of an c5n.18xlarge instance for a main node and 40 c5.4xlarge instances as worker nodes. These instances have respectively 36 and 8 2-way hyperthreaded Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz cores with respectively 196 GB and 32 GB of RAM.

Each of our worker nodes only requires 2 GB of RAM because sketch deltas are small and workers are stateless. However, AWS workers with sufficient CPU power come with more RAM than we need.

A note about experimental comparisons. Ideally, we would include experimental comparisons against existing distributed systems that solve connectivity or k-connectivity on graph streams with edge insertions and deletions. However, only one such system (KickStarter [74]) exists in the literature, and its source code is not available (see Appendix G). Instead, we compare against a theoretical upper bound for stream ingestion: the data acquisition cost.

We define the ***data acquisition cost*** to be the cost of receiving the input stream at the main node over a network link and logging it in RAM. Sequential RAM bandwidth is a trivial bound on this cost.

- ***RAM sequential bandwidth*** is the maximum rate at which the CPU can write consecutive words in RAM on any number of threads.
- ***RAM random access bandwidth*** is the maximum worst-case rate at which the CPU can write any sequence of words in RAM on any number of threads.

Experiment Metrics. Our experiments process the entire stream of updates and then perform a single query at the end of the stream. Where specified experiments also perform additional queries throughout the processing of the stream.

We report the update throughput by measuring wall-clock time from the beginning of the stream until all updates have been applied to the sketches. We also measure the total amount of network communication to/from the main node and the amount of RAM usage on the main node.

When performing a query we measure the wall-clock latency from the moment the query is issued to the time the answer is returned to the user. This time includes the latency of flushing all pending updates from the pipeline hypertree and then the query computation itself.

7.1 Datasets In many of the experiments below, we use the synthetically generated graph streams used in the evaluation of GRAPHZEPPELIN [71]. These graphs were generated using the Graph500 Kronecker graph generator specification [4], and are very dense: each graph contains approximately 1/4 of all possible edges.

For larger-scale experiments, we evaluate LANDSCAPE on randomly generated Erdos-Renyi graphs (with 2^{18}, 2^{19}, and 2^{20} vertices) with edge probability set to 1/4.

Finally, we evaluate LANDSCAPE on real-world graph datasets from the SNAP graph repository [47] and NetworkRepository [65].

All of the above graphs were transformed into a random streams of edge insertions and deletions using the method described in [71]. We additionally inserted and removed all edges seven times to increase stream length. Each random stream, once insert/delete pairs for the same edge are removed, is exactly the edge list of the graph used to generate it. These datasets are summarized in Table 2.

7.2 Landscape is Highly Scalable. We measured LANDSCAPE's stream ingestion rate and network bandwidth usage on kron17 given varying numbers of distributed

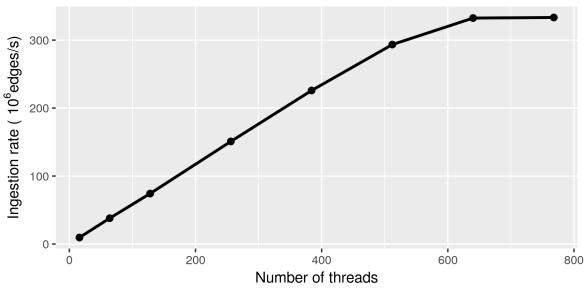


Figure 3: LANDSCAPE ingestion rate scales to one-fourth of sequential RAM bandwidth.

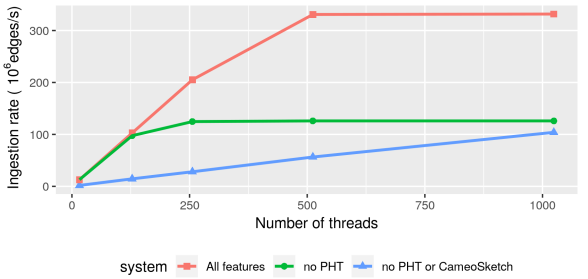


Figure 4: CAMEOSKETCH and pipeline hypertree are vital for good ingestion performance. Without CAMEOSKETCH, LANDSCAPE's ingestion rate scales slowly as the number of threads increases. Without pipeline hypertree, the system bottlenecks at slightly over 100 million updates/sec.

workers. Figure 3 demonstrates a near-linear increase in ingestion rate as more distributed threads are added, until throughput levels off as it approaches 340 million updates/sec. The ingestion rate on one worker node (with 16 threads) is about 9.6 million updates/sec, and the ingestion rate on 40 worker nodes (with a total of $40 \cdot 16 = 640$ threads) is 332 million updates/sec, a 35× speedup. We note that at this point LANDSCAPE is observably ***not*** CPU-bound: adding more worker nodes no longer increases throughput, and we measure instructions per cycle on the main node CPU to be 0.8, indicating that the CPU is not instruction bound, but RAM bandwidth bound [28].

Since graph stream updates are 9 bytes, LANDSCAPE ingestion bandwidth (2.8 GiB/sec) is four times that of random-access RAM bandwidth (759 MiB/sec), and roughly one-fourth of sequential RAM access bandwidth (12.4 GiB/sec) on the (c5n.18xlarge) main node.

Throughout these tests we observe a constant amount of network communication used; approximately 1.6 times the size of the input stream for dense graphs. See Table 5 for a more complete evaluation.

Performance impact of CameoSketch and pipeline hypertree. Figure 4 illustrates the performance impact of CAMEOSKETCH and the pipeline hypertree on LANDSCAPE's ingestion rate with the kron17 dataset. When LANDSCAPE uses GRAPHZEPPELIN's buffering data structure and its CUBESKETCH sketch algorithm, its ingestion rate increases as more distributed workers are added, but at a slow rate. Using CAMEOSKETCH alongside GRAPHZEPPELIN's buffering system results in a much faster increase in ingestion rate, but the system bottlenecks at roughly 120 million updates/sec. In contrast, the full LANDSCAPE system continues to increase its ingestion rate dramatically as more workers are added, and bottlenecks at over 300 million updates/sec. We conclude that the $O(\log V)$ decrease in ingestion cost for CAMEOSKETCH and the improved design of the pipeline hypertree are vital for LANDSCAPE's performance. See Appendix F.4 for a direct experimental comparison of LANDSCAPE to GRAPHZEPPELIN on a single machine.

Table 3: LANDSCAPE has a high ingestion rate on sufficiently dense graphs and has low communication overhead.

Dataset	Ingestion Rate ($\times 10^6$ updates/sec)	Communication (as a factor of stream size)
kron13	231	1.6
kron15	336	1.6
kron16	334	1.6
kron17	335	1.6
ca-citeseer	17.3	1.7
p2p-gnutella	13.5	0
rec-amazon	12.5	0
google-plus	134	2.6
web-uk-2005	91.5	3.4
erdos18	291	1.6
erdos19	226	1.6
erdos20	236	1.6

More datasets. Table 3 summarizes LANDSCAPE's stream ingestion rate and network costs using 640 worker threads on a variety of synthetic and real-world datasets. Its ingestion rate is very high on dense graph streams and on real-wold streams google-plus and web-uk-2005. It is lower on ca-citeseer, p2p-gnutella and rec-amazon because these datasets do not contain enough stream updates to surpass LANDSCAPE's 4% leaf fullness threshold. So, for these streams, a large portion (or all) of update processing occurs on the main node.

Circumventing bottlenecks. These experiments support our claims that sketching can avoid the traditional bottlenecks in distributed graph stream processing. They demonstrate that CPU cost can be distributed away, supporting Claim 1.2; that network communication can be only a constant factor of the data acquisition cost, supporting Claim 1.3; and that stream processing

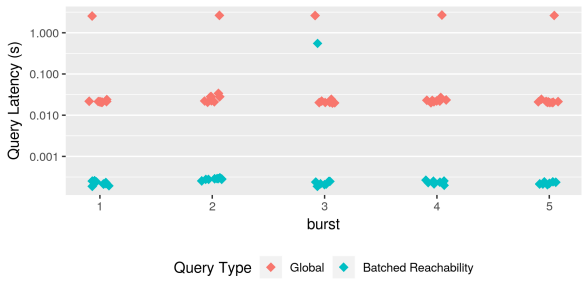


Figure 5: GreedyCC dramatically decreases query latency.

Table 4: When computing k-connectivity, increasing k incurs a linear decrease in ingestion rate, a linear increase in sketch size, a quadratic increase in query latency, and does not affect total network communication. Reported values obtained by running Landscape on the kron17 dataset.

	Ingestion Rate (10^6 u/s)	Sketch Size (GiB)	Query (seconds)	Network (GiB)
$k = 1$	338.5	15.25	1.27	425.2
$k = 2$	200	24.40	5.02412	464.981
$k = 4$	101.5	46.49	16.1608	468.886
$k = 8$	50.76	83.39	65.5716	476.598

bandwidth can reach to within a factor 4 of sequential RAM bandwidth, supporting Claim 1.4.

7.3 Landscape Answers Queries Quickly. We measured Landscape's query latency for both global connectivity queries and batched reachability queries on an extended kron17 stream. For batched reachability queries, we uniformly sample the pairs to query $(v_1, v_2), (v_3, v_4), \ldots$ from the set of all vertices. We issue queries periodically and record (1) the time to flush the pipeline hypertree and update the graph sketch and (2) to perform Borůvka's algorithm using the graph sketch. The sum of these times is the total query latency, though only the Borůvka computation is additional work induced by the query; the flushing work would have happened anyway as part of stream ingestion even if the query was never issued. We found that flushing takes roughly 2.3 seconds, while Borůvka's algorithm takes 0.3 seconds.

We also evaluate how query latency and ingestion rate are affected by the use of GreedyCC. We measure the performance impact of this optimization by running Landscape on extended kron17 and issuing ***query bursts***: multiple connectivity queries issued close together (within 50 thousand stream updates). We track the latency of each query in the burst. The results are summarized in Figure 5. The first query in each burst, which requires flushing and running Borůvka's algorithm to answer, has high (multi-second) latency. However, we see that for the remaining queries in the burst the latency is much lower: two orders of magnitude lower for global connectivity queries, and up to four orders of magnitude lower for batched reachability queries.

7.4 k-connectivity Performance We repeat our scaling, network communication, and query latency experiments on the k-connectivity problem for a variety of datasets and values of k. Table 4 highlights Landscape's performance for computing k-connectivity on the kron17 dataset. Note that increasing k incurs a linear decrease in ingestion rate, a linear increase in sketch size, a quadratic increase in query latency, and has no significant effect on network communication. These experimental results support the asymptotic conclusions in Theorem 5.4. See Appendix F.3 for a summary of Landscape's performance on more datasets.

8 Conclusion

This paper demonstrates how to use linear sketching to avoid the space, CPU & network bottlenecks faced by existing graph processing systems. We make our argument in the context of the Landscape system for finding connected components of dense, dynamic graphs.

By avoiding these bottlenecks, Landscape achieves remarkable performance. Specifically, it supports a stream ingestion rate one-fourth of sequential RAM bandwidth.

9 Appendix

The text of the appendix can be found in the full version of this paper[2].

Acknowledgments

This work was supported in part by NSF grants CCF-2247577, CCF-2106827, and NRT-HDR 2125295.

References

[1] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 459–467. SIAM, 2012.

[2] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *PODS*, pages 5–14. ACM, 2012.

[3] Reka Albert. Scale-free networks in cell biology. *Journal of cell science*, 118(21):4947–4957, 2005.

[4] J. Ang, Brian W. Barrett, Kyle B. Wheeler, and Richard C. Murphy. Introducing the graph 500. 2010.

[2] Full paper: https://arxiv.org/abs/2410.07518

[5] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[6] Sepehr Assadi and Vihan Shah. Tight bounds for vertex connectivity in dynamic streams. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 213–227. SIAM, 2023.

[7] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming B-trees. In *Proc. 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 81–92, 2007.

[8] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. An introduction to b^ε-trees and write-optimization. *:login; magazine*, 40(5):22–28, October 2015.

[9] Tanya Y. Berger-Wolf and Jared 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.

[10] Ilaria Bordino and Debora Donato. Dynamic characterization of a large web graph.

[11] Federico Busato, Oded Green, Nicola Bombieri, and David A Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[12] Deng Cai, Zheng Shao, Xiaofei He, Xifeng Yan, and Jiawei Han. Mining hidden community in heterogeneous social networks. In *Proceedings of the 3rd International Workshop on Link Discovery*, LinkKDD '05, page 58–65, New York, NY, USA, 2005. Association for Computing Machinery.

[13] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 85–98, New York, NY, USA, 2012. Association for Computing Machinery.

[14] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 85–98, 2012.

[15] Yann Collet. xxhash-extremely fast non-cryptographic hash algorithm. *URL https://github.com/Cyan4973/xxHash*, 2016.

[16] Graham Cormode and Donatella Firmani. A unifying framework for ℓ_0-sampling algorithms. *Distributed and Parallel Databases*, 32, 09 2014.

[17] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[18] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[19] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014)*. USENIX – Advanced Computing Systems Association, April 2014.

[20] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew J. Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[21] David Ediger, Rob McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5. IEEE, 2012.

[22] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.

[23] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. Effective community search for large attributed graphs. *Proc. VLDB Endow.*, 9(12):1233–1244, August 2016.

[24] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. Effective community search for large attributed graphs. *Proc. VLDB Endow.*, 9(12):1233–1244, August 2016.

[25] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2):207–216, December 2005.

[26] Guoyao Feng, Xiao Meng, and Khaled Ammar. Distinger: A distributed graph data structure for massive dynamic graph processing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 1814–1822, 2015.

[27] Evangelos Georganas, Rob Egan, Steven Hofmeyr, Eugene Goltsman, Bill Arndt, Andrew Tritt, Aydin Buluç, Leonid Oliker, and Katherine Yelick. Extreme scale de novo metagenome assembly. SC '18. IEEE Press, 2018.

[28] Brendan Gregg. The pmcs of ec2: Measuring ipc, May 2017.

[29] Sudipto Guha, Andrew McGregor, and David Tench. Vertex and hyperedge connectivity in dynamic graph streams. In *PODS*, pages 241–247. ACM, 2015.

[30] Erez Hartuv and Ron Shamir. A clustering algorithm based on graph connectivity. *Information Processing Letters*, 76(4):175–181, 2000.

[31] Lifeng He, Yuyan Chao, Kenji Suzuki, and Kesheng Wu. Fast connected-component labeling. *Pattern recognition*, 42(9):1977–1987, 2009.

[32] Lifeng He, Xiwei Ren, Qihang Gao, Xiao Zhao, Bin Yao, and Yuyan Chao. The connected-component labeling problem: A review of state-of-the-art algorithms. *Pattern Recognition*, 70:25–43, 2017.

[33] M. Moftah Hossam, Aboul Ella Hassanien, and Mohamoud Shoman. 3d brain tumor segmentation scheme using k-mean clustering and connected component labeling

algorithms. In *2010 10th International Conference on Intelligent Systems Design and Applications*, pages 320–324, 2010.

[34] Anand Iyer, Li Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. pages 1–6, 06 2016.

[35] Anand Iyer, Li Erran Li, and Ion Stoica. Celliq: Real-time cellular network analytics at scale. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 309–322, 2015.

[36] Anand Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. Tegra: Efficient ad-hoc analytics on evolving graphs. In *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, April 2021.

[37] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *Proceedings of the fourth international workshop on graph data management experiences and systems*, pages 1–6, 2016.

[38] Jinhong Jung, Kijung Shin, Lee Sael, and U Kang. Random walk with restart on large graphs using block elimination. *ACM Transactions on Database Systems (TODS)*, 41(2):1–43, 2016.

[39] U. Kang and Christos Faloutsos. Beyond 'caveman communities': Hubs and spokes for graph compression and mining. pages 300–309, 12 2011.

[40] U. Kang, Mary McGlohon, Leman Akoglu, and Christos Faloutsos. Patterns on the connected components of terabyte-scale graphs. pages 875–880, 12 2010.

[41] Goossen Kant. *Algorithms for drawing planar graphs*. PhD thesis, 1993.

[42] Michael Kapralov, Yin Tat Lee, Cameron Musco, Christopher Musco, and Aaron Sidford. Single pass spectral sparsification in dynamic streams. In *FOCS*, pages 561–570. IEEE Computer Society, 2014.

[43] David R Karger. A randomized fully polynomial time approximation scheme for the all terminal network reliability problem. In *Proceedings of the twenty-seventh annual ACM Symposium on Theory of Computing*, pages 11–17, 1995.

[44] Michael Korn, Daniel Sanders, and Josef Pauli. Moving object detection by connected component labeling of point cloud registration outliers on the gpu. In *VISIGRAPP (6: VISAPP)*, pages 499–508, 2017.

[45] Krishnamurthy. An improved min-cut algonthm for partitioning vlsi networks. *IEEE Transactions on computers*, 100(5):438–446, 1984.

[46] Wookey Lee, James J Lee, and Jinho Kim. Social network community detection using strongly connected components. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 596–604. Springer, 2014.

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

[48] Yongsub Lim, U Kang, and Christos Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):3077–3089, 2014.

[49] Yongsub Lim, Won-Jo Lee, Ho-Jin Choi, and U Kang. Discovering large subsets with high quality partitions in real world graphs. In *2015 International Conference on Big Data and Smart Computing (BIGCOMP)*, pages 186–193. IEEE, 2015.

[50] Yongsub Lim, Won-Jo Lee, Ho-Jin Choi, and U Kang. Mtp: discovering high quality partitions in real world graphs. *World Wide Web*, 20(3):491–514, 2017.

[51] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*, pages 363–374. IEEE, 2015.

[52] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[53] Andrew McGregor. Graph stream algorithms: a survey. *ACM SIGMOD Record*, 43(1):9–20, 2014.

[54] Duccio Medini, Antonello Covacci, and Claudio Donati. Protein homology network families reveal step-wise diversification of type iii and type iv secretion systems. *PLoS computational biology*, 2(12):e173, 2006.

[55] Derek G Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. Incremental, iterative data processing with timely dataflow. *Communications of the ACM*, 59(10):75–83, 2016.

[56] Zachary P Neal, Rachel Domagalski, and Bruce Sagan. Comparing alternatives to the fixed degree sequence model for extracting the backbone of bipartite projections. *Scientific reports*, 11(1):23929, 2021.

[57] Sergey Nurk, Dmitry Meleshko, Anton Korobeynikov, and Pavel Pevzner. Metaspades: A new versatile metagenomic assembler. *Genome Research*, 27:gr.213959.116, 03 2017.

[58] RS Olson and ZP Neal. Navigating the massive world of reddit: Using backbone networks to map user interests in social media. arxiv. org. *URL: http://arxiv. org/abs/1312.3387*, 2013.

[59] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

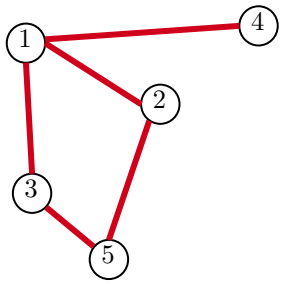
[60] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1372–1385, 2021.

[61] Md. Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. A new scalable parallel dbscan algorithm using the disjoint-set data structure. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012.

[62] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. 16(4):303–324, December 1990.

[63] Shafiur Rahman, Mahbod Afarin, Nael Abu-Ghazaleh, and Rajiv Gupta. Jetstream: Graph analytics on streaming data with event-driven hardware accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1091–1105, New York,

NY, USA, 2021. Association for Computing Machinery.
[64] Aparna Ramanathan and Charles J Colbourn. Counting almost minimum cutsets with reliability applications. *Mathematical Programming*, 39:253–261, 1987.
[65] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.
[66] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, December 2017.
[67] Russell Sears, Mark Callaghan, and Eric A. Brewer. *Rose*: compressed, log-structured replication. *Proc. VLDB Endow.*, 1(1):526–537, 2008.
[68] Dipanjan Sengupta and Shuaiwen Leon Song. Evograph: On-the-fly efficient mining of evolving graphs on gpu. In *International Supercomputing Conference*, pages 97–119. Springer, 2017.
[69] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. Graphin: An online high performance incremental graph processing framework. In *European Conference on Parallel Processing*, pages 319–333. Springer, 2016.
[70] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, mar 1984.
[71] David Tench, Evan West, Victor Zhang, Michael A. Bender, Abiyaz Chowdhury, J. Ahmed Dellas, Martin Farach-Colton, Tyler Seip, and Kenny Zhang. Graphzeppelin: Storage-friendly sketching for connected components on dynamic graph streams. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 325–339, New York, NY, USA, 2022. Association for Computing Machinery.
[72] Heidi Thornquist, Eric Keiter, Robert Hoekstra, David Day, and Erik Boman. A parallel preconditioning strategy for efficient transistor-level circuit simulation. pages 410–417, 01 2009.
[73] S van Dongen. Graph clustering by flow simulation. Technical report, 2000.
[74] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 237–251, New York, NY, USA, 2017. Association for Computing Machinery.
[75] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. Aspire: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, page 861–878, New York, NY, USA, 2014. Association for Computing Machinery.
[76] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. Efficient structural graph clustering: An index-based approach. *The VLDB Journal*, 28(3):377–399, June 2019.
[77] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. Efficient structural graph clustering: An index-based approach. *The VLDB Journal*, 28(3):377–399, June 2019.
[78] Min Wu, Xiaoli li, Chee-Keong Kwoh, and See-Kiong Ng. A core-attachment based method to detect protein complexes in ppi networks. *BMC bioinformatics*, 10:169, 02 2009.
[79] Z. Wu and R. Leahy. An optimal graph theoretic approach to data clustering: theory and its application to image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(11):1101–1113, 1993.
[80] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, and Divesh Srivastava. Automatic discovery of attributes in relational databases. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 109–120, New York, NY, USA, 2011. Association for Computing Machinery.
[81] Tobias Ziegler, Carsten Binnig, and Viktor Leis. Scalestore: A fast and cost-efficient storage engine using dram, nvme, and rdma. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 685–699, New York, NY, USA, 2022. Association for Computing Machinery.

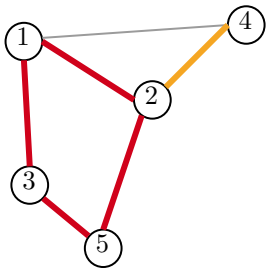


(a) Graph $\mathcal{G}$.

	(1,2)	(1,3)	(1,4)	(1,5)	(2, 3)	(2,4)	(2,5)	(3,4)	(3,5)	(4,5)
1	1	1	0	0	0	0	0	1	1	1
2	1	0	0	0	0	1	1	0	0	0
3	0	1	0	0	0	0	0	0	1	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	1	1	0	0	0

(b) $\mathcal{G}$'s characteristic vectors.

Figure 6: An example graph $\mathcal{G}$ and its characteristic vectors.



(a) Graph $\mathcal{G}$.

	(1,2)	(1,3)	(1,4)	(1,5)	(2, 3)	(2,4)	(2,5)	(3,4)	(3,5)	(4,5)
1	1	1	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	1	0	0	0
3	0	1	0	0	0	0	0	0	0	1
4	0	0	1	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1	0	0	1

(b) $\mathcal{G}$'s characteristic vectors. The values changed by the edge updates are indicated in red.

Figure 7: An example of updating graph $\mathcal{G}$ and its characteristic vectors with an orange edge insertion $(2,4)$ and a grey edge deletion $(1,4)$.

A More Detail on Ahn et al.'s [1] algorithm

Recall the definition of a ***characteristic vector*** from Section 4: $f_u \in \mathbb{Z}_2^{\binom{V}{2}}$ of each vertex $u \in \mathcal{V}$ such that each nonzero element of f_u denotes an edge incident to u. That is, $f_u \in \mathbb{Z}_2^{\binom{V}{2}}$ s.t. for all vertices $0 \leq i < j < V$:

$$f_u[(i,j)] = \left\{ \begin{array}{ll} 1 & u \in \{i,j\} \text{ and } (i,j) \in \mathcal{E} \\ 0 & \text{otherwise} \end{array} \right\}$$

First, we give an example of how one can represent a graph as a set of characteristic vectors, how these vectors can be updated as edges are inserted or deleted, and how they can be used to compute connected components using Borůvka's algorithm. Then, we outline how Ahn et al. use linear sketching to reduce the space cost of this approach to $O(V \log^3 V)$.

Consider the graph $\mathcal{G}$ its the characteristic vectors $\{f_i \mid i \in [V]\}$ for each of its nodes as shown in Figure 6.

When we receive an edge insertion or deletion for an edge (u,v), we can update the characteristic vectors f_u and $f_v \in \mathbb{Z}_2^{\binom{V}{2}}$ by adding the one-hot vector $e_{(u,v)}$ to each of them. Specifically, $e_{(u,v)}$ is a vector with a 1 in the position corresponding to the edge (u,v) and 0 in all other positions. Note that for both insertions and deletions, we update the characteristic vectors in the same manner.

For instance, imagine that we stream in one edge insertion of $(2,4)$ and one edge deletion of $(1,4)$.

For the **insertion** of the edge $(2,4)$, we add a one-hot vector $e_{(2,4)}$ to the rows f_2 and f_4. Likewise, for the **deletion** of $(1,4)$, we can add the vector $e_{(1,4)}$ to the rows f_1 and f_4. Adding those one-hot vectors $e_{(1,4)}$ and $e_{(2,4)}$ to the appropriate characteristic vectors updates our representation of $\mathcal{G}$ to reflect these updates.

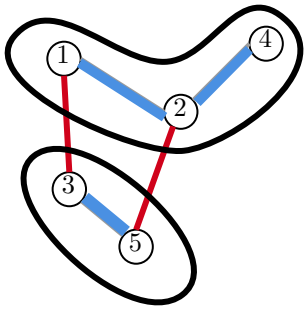
The process of performing these edge updates and their impact upon the graph $\mathcal{G}$ and characteristic vectors is shown in Figure 7.

Computing the connected components. Recall that Borůvka's algorithm samples an edge from each node, merges the nodes at the endpoints of every sampled edge into supernodes, and repeats this procedure either until there only remain supernodes with no edges between them (which is, in the worst-case, after $\log_2 V$ rounds have passed).

We can easily do this with the characteristic vectors: From each characteristic vector, we choose an arbitrary nonzero entry. In this round, it so happens that 1 samples $(1,2)$, 2 samples $(1,2)$, 3 samples $(3,5)$, 4 samples $(2,4)$, and 5 samples $(3,5)$. We indicate the sampled edges in blue on Figure 8.

To merge the components that were newly found to be connected by sampled edges, we sum the characteristic vectors of the nodes that were merged. In general, if we sampled an edge (u,v), then the merged supernode for u and v will have a characteristic vector f_{uv} that is equal to $f_u + f_v$ (recall that our vectors are over $\mathbb{Z}_2$, and therefore this operation generates a vector corresponding to the symmetric difference of the edges adjacent to u and v, which is exactly the edges out of u and v that don't have one endpoint in u and another in v).

In this example (shown in Figure 8), we find edges connecting $\{1,2,4\}$ and $\{3,5\}$. Thus, we set $f_{124} = f_1 + f_2 + f_4$

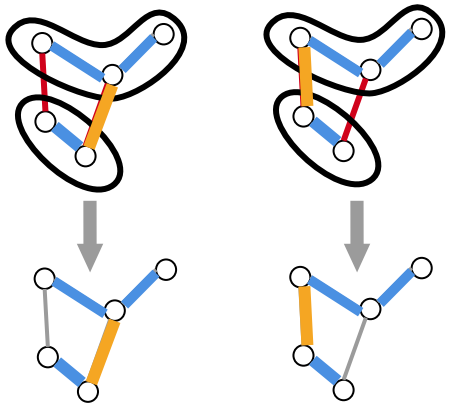


(a) Graph of the supernodes we obtain after the round. Supernodes are indicated in purple with the edges we found in blue and the edges yet to be found in red.

	(1,2)	(1,3)	(1,4)	(1,5)	(2, 3)	(2,4)	(2,5)	(3,4)	(3,5)	(4,5)
1+2+4	0	1	0	0	0	0	1	0	0	0
3+5	0	1	0	0	0	0	1	0	0	0

(b) The supernodes' merged characteristic vectors.

Figure 8: An example of running a round of Borůvka's algorithm. After sampling the blue edges we merge the characteristic vectors to achieve representations of the supernodes (in black).



(a) The two possible spanning forests we can obtain in the final round. We can sample either of the two edges indicated in orange to obtain two different spanning trees.

	(1,2)	(1,3)	(1,4)	(1,5)	(2, 3)	(2,4)	(2,5)
(1+2+4)+(3+5)	0	0	0	0	0	0	0

(b) The characteristic vector of the final supernode.

Figure 9: An example of running the final round of Borůvka's algorithm. We can sample either of the orange edges but obtain the same supernode either way.

and $f_{35} = f_3 + f_5$. Note that these new characteristic vectors retain the edges ($(1,3)$ and $(2,5)$) that cross between supernodes.

Finally, in the next round (shown in Figure 9), both the supernode representing $\{1,2,4\}$ and the supernode representing $\{3,5\}$ may sample either of the edges $(1,3)$ and $(2,5)$. Depending on which edge we sample, we can get a different spanning forest. In either case, there is now a single supernode for our connected component, with the following edge incidence indicator vector resulting from summing the two remaining rows.

Thus, storing the characteristic vectors for each node, and updating them on edge insertions and deletions, allows us to compute connected components using Borůvka's algorithm. Storing these characteristic vectors in a streaming context, however is infeasible. Each characteristic vector has $O(V^2) = O(V^2)$ bits, and thus the total space required to store all of them is $O(V^3)$ bits.

Sketching the Characteristic Vectors The trick Ahn et al. use to make this approach space-efficient is to maintain several ℓ_0-sketches of each characteristic vector rather than represent the vector losslessly. Recall from Theorem 4.1 that each such sketch has size $O(\log^2 V)$ because δ (the probability of sketch failure) is set to a constant. Since an ℓ_0-sketch is a linear function, we can see that adding the sketches of two characteristic vectors is equivalent to first adding the vectors, and then sketching the result: $\mathcal{S}(f_u + f_v) = \mathcal{S}(f_u) + \mathcal{S}(f_v)$. A similar result holds for updating a vector after an edge insertion or deletion. So we can perform all the same operations on the sketches that we would on the characteristic vectors, and get the same result (with the exception that querying a sketch for an edge during Borůvka's algorithm fails with constant probability).

We maintain $O(\log V)$ ℓ_0-sketches for each characteristic vector. It is straightforward to prove that this is sufficient to find all the connected components with high probability. We describe several specific ℓ_0-sketching algorithms in the next section.

B Sketching

In this section we describe the design of ℓ_0-sketches (including our new sketch CameoSketch) and how these sketches can be used to compute graph connectivity.

B.1 ℓ_0-Sketching

Problem 3. (ℓ_0-sampling) *A vector x of length n is defined by an input stream of updates of the form (i, Δ) where value Δ is added to x_i, and the task is to sample a nonzero element of x using $o(n)$ space.*

We denote the ℓ_0 sketch of a vector x as $\mathcal{S}(x)$. The sketch is a linear function, i.e., $\mathcal{S}(x) + \mathcal{S}(y) = \mathcal{S}(x + y)$ for any vectors x and y. Work by Cormode et al. established

```
1: function UPDATE_SKETCH(idx) ▷ Toggle vector index 'idx'
2:     for all col ∈ [0, log(1/δ)) do
3:         col_hash ← hash_1(col, idx)
4:         row ← 0
5:         checksum ← hash_2(col, idx)
6:         while row == 0 OR col_hash[row-1] == 0 do
7:             b_{row,col}.α ← b_{row,col}.α ⊕ idx
8:             b_{row,col}.γ ← b_{row,col}.γ ⊕ checksum
9:             row ← row + 1
10: function QUERY_SKETCH( ) ▷ Get a non-zero vector index
11:     for all col ∈ [0, log(1/δ)) do
12:         for all row ∈ [0, log(V)) do
13:             if b_{row,col}.γ == hash_2(col, b_{row,col}.α) then
14:                 return b_{row,col}α ▷ Found a good bucket, done
15:     return sketch_failure ▷ All buckets bad
```

Figure 10: Pseudocode for GRAPHZEPPELIN's CUBESKETCH. Each edge update (u, v) is converted to and from a vector index for use with the sketch.

that there is an ℓ_0-sampler that succeeds with probability at least $1 - \delta$ and uses $O(\log^2(n)\log(1/\delta))$ bits of space.

Previous work by Tench et al. [71] introduced the CUBESKETCH algorithm, an improved ℓ_0-sampler for the special case of vectors $\in \mathbb{Z}_2^n$ which we describe below:

B.2 CubeSketch. Given a vector $f \in \mathbb{Z}_2^n$, a CUBESKETCH consists of a matrix of $\log(n)$ by $\log(1/\delta)$ ***buckets***. Each bucket lossily represents the values at a random subset of positions of f. It does so with two values $\boldsymbol{b.\alpha}$, which is the xor of all nonzero positions, and $\boldsymbol{b.\gamma}$, the xor of the hash of each nonzero position. We can recover a nonzero element of f from bucket $b_{i,j}$ only when a single position in $b_{i,j}$ is nonzero. In this case we call bucket $b_{i,j}$ ***good***, otherwise we say it is ***bad***. When b is good, then $b.\alpha$ is equal to the nonzero index and $\text{hash}(b.\alpha) = b.\gamma$. If b is bad then with high probability $\text{hash}(b.\alpha) \neq b.\gamma$.

The procedure for updating index i is outlined in Figure 10; for each column c, row r is chosen with probability $1/2^r$ (using a 2-wise independent hash function $h_j : [n] \to \log n$) and i is applied to each $b_{j,c}$ for $j \leq r$. See Figure 11 for an example of CUBESKETCH's update procedure.

B.3 Landscape's new sketch: CameoSketch Essential to the performance LANDSCAPE achieves is our new ℓ_0-sampler called CAMEOSKETCH. CAMEOSKETCH improves over CUBESKETCH with a new update procedure that is a factor $\log n$ faster to update and reduces space usage by a constant factor via a refined analysis. All other details, including the query procedure, remain unchanged.

Update procedure. CAMEOSKETCH maintains the same matrix of buckets as CUBESKETCH, but uses a simpler and faster update procedure as shown in Figure 12. When performing an update u, for each column c we choose

$\{1,4,7,9,11\}$	$\{1,4,7,9,11\}$	$\{1,4,7,9,11\}$
$\{1,4,11\}$	$\{1,9,11\}$	$\{1,4,7,9,11\}$
$\{1\}$	$\{9,11\}$	$\{1,7,9,11\}$
$\{\}$	$\{\}$	$\{1,9\}$

0	1	0	0	1	0	0	0	0	1	0	1

(a) CUBESKETCH buckets (above) and the input vector (below) after adding 1 to index 9.

$\{\cancel{1},4,7,9,11\}$	$\{\cancel{1},4,7,9,11\}$	$\{\cancel{1},4,7,9,11\}$
$\{\cancel{1},4,11\}$	$\{\cancel{1},9,11\}$	$\{\cancel{1},4,7,9,11\}$
$\{\cancel{1}\}$	$\{9,11\}$	$\{\cancel{1},7,9,11\}$
$\{\}$	$\{\}$	$\{\cancel{1},9\}$

0	0	0	0	1	0	0	0	0	1	0	1

(b) CUBESKETCH buckets (above) and the input vector (below) after adding 1 to index 1.

Figure 11: Example of the structure of CUBESKETCH buckets. In this example, the sketch has 3 columns and 4 rows of buckets. Toggling a vector index either sets it to 1 (and thus adds it to the buckets) or sets it to 0 (thus removing it from the buckets). In this example we begin with nonzero indices $\{1, 4, 7, 11\}$. We then flip index 9 to 1 and then flip index 1 to 0. We can extract nonzero indices from any good buckets. That is, buckets which contain only one nonzero index.

```
1: function UPDATE_SKETCH(idx) ▷ Toggle vector index 'idx'
2:     checksum ← hash_2(idx)
3:     for all col ∈ [0, log(1/δ)] do
4:         depth ← log_2(hash_1(col, idx))
5:         b_0.α ← b_0.α ⊕ idx
6:         b_0.γ ← b_0.γ ⊕ checksum
7:         b_{depth}.α ← b_{depth}.α ⊕ idx
8:         b_{depth}.γ ← b_{depth}.γ ⊕ checksum
```

Figure 12: Pseudocode for LANDSCAPE's CAMEOSKETCH update procedure. Each edge update (u, v) is converted to and from a vector index for use with the sketch.

a row r independently with probability $1/2^r$. Unlike CUBESKETCH which updates all rows $[0, r]$ in the column, we apply u to only $b_{0,c}$ and $b_{r,c}$.

It is straightforward to see that CAMEOSKETCH maintains the correctness and probabilistic bounds of CUBESKETCH. Assuming they use the same hash functions, if a bucket is good in the latter it must also be good in the former.

Recall THEOREM 4.2 CAMEOSKETCH *is an ℓ_0-sampler that, for vector $x \in \mathbb{Z}_2^n$, uses $O(\log^2(n)\log(1/\delta))$ space, has worst-case update time $O(\log(1/\delta))$, and succeeds with probability $1 - \delta$.*

$\{1,4,7,9,11\}$	$\{1,4,7,9,11\}$	$\{1,4,7,9,11\}$
$\{4,11\}$	$\{1\}$	$\{4\}$
$\{1\}$	$\{9,11\}$	$\{7,11\}$
$\{\}$	$\{\}$	$\{1,9\}$

0	1	0	0	1	0	0	0	0	1	0	1

(a) CameoSketch buckets (above) and the input vector (below) after adding 1 to index 9.

$\{\cancel{1},4,7,9,11\}$	$\{\cancel{1},4,7,9,11\}$	$\{4,7,9,11\}$
$\{4,11\}$	$\{\cancel{1}\}$	$\{4\}$
$\{\cancel{1}\}$	$\{9,11\}$	$\{7,11\}$
$\{\}$	$\{\}$	$\{\cancel{1},9\}$

0	0	0	0	1	0	0	0	0	1	0	1

(b) CameoSketch buckets (above) and the input vector (below) after adding 1 to index 1.

Figure 13: Example of the structure of CameoSketch buckets. The setup for this example is the same as for CubeSketch (see Figure 11). However, in CameoSketch we update fewer buckets for each index. As we can see in this figure only 2 buckets per column contain a given index. The result is an asymptotic improvement to update time and an increase in the number of good buckets.

Proof. The proof follows from the analysis of CubeSketch. If a CubeSketch and CameoSketch share the same randomness, then the CubeSketch returning a valid edge implies that the CameoSketch must also return a valid edge. This is because each CameoSketch bucket contains a subset of the contents of the same CubeSketch bucket. If the CubeSketch column is good then there exists a deepest (largest row value) good bucket $b_{i,j}$. CameoSketch's $b_{i,j}$ must be identical to CubeSketch's. This is because each nonzero index in CameoSketch appears at its deepest row and because of the subset property.

The only exception to this property is if CameoSketch returns an incorrect edge due to a checksum error. However, this happens with polynomially small probability and thus does not violate the proof. □

Theorem 4.2 demonstrates that CameoSketch reduces the CPU burden of performing updates and supports Claim 1.2.

Reduced constant factors. In GraphZeppelin, Tench et al. use $72\log(1/\delta)\log n$ bytes of space to guarantee a failure probability of at most δ when sketching a vector of length $n < 2^{64}$ using CubeSketch. Via a careful constant-factor analysis, we can show that CameoSketch can match this failure probability with significantly less space:

Recall Theorem 4.3. *Using 3-wise independent hash functions,* CameoSketch *requires* $8\log_3(1/\delta)(\log n + 5)$ *bytes of space to return a nonzero element of a length* $n < 2^{64}$ *input vector w/p at least* $1-\delta$.

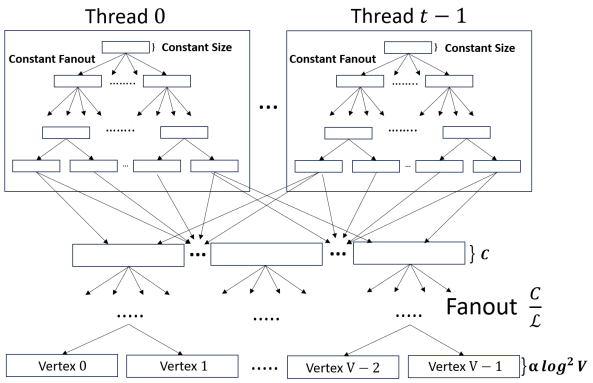


Figure 14: The Pipeline Hypertree data-structure.

The intuition behind this result is that for every sketch column there exists a depth d such that fewer than 7 of the stored elements exist below it. Given this knowledge, we can use fully independent analysis upon this subset of the buckets to achieve a tighter lower bound on the probability of success. We omit the complete proof due to space constraints.

Theorem 4.3 immediately implies a space savings of up to 90% compared to CubeSketch and thus supports Claim 1.1. In our implementation, we conservatively choose to use slightly more space than this theorem requires to reduce the failure probability further. Still, our implementation requires only 2/7ths of the space used in GraphZeppelin [71] (see Section 6 for details).

C Design of Pipeline Hypertree

To make vertex-based batching fast, we design the ***pipeline hypertree***, which is a simplified and parallel variant of the buffer tree [5] designed to minimize cache line misses and thread contention. The pipeline hypertree receives arbitrarily ordered stream updates and consolidates them into vertex-based batches.

The pipeline hypertree is logically structured as a directed acyclic graph (DAG) D with $O(V+t)$ nodes and $O(V+t)$ edges, managed by t threads. The nodes are partitioned into $O(\log_{C/\mathcal{L}} V)$ ***levels***, where C denotes the size of L3 cache and $\mathcal{L}$ denotes the size of an L3 cache line. Nodes of D in levels 0 through ρ (for some constant ρ) are ***thread local***: for each node in level $l \leq \rho$ there is exactly one thread that can read from or write to the node.

Specifically, there are t nodes in level 0 and each thread owns a unique level 0 node. Nodes in levels 0 to $\rho - 1$ each have constant y children, and each thread has exclusive access to all descendants of a level 0 node (up to level ρ). For levels $\geq \rho$, each node has $O(C/\mathcal{L})$ children. These

global nodes in levels $> \rho$ can be read from or written to by any thread. At the lowest level of D, there are V leaf nodes, one for each vertex in the input graph $\mathcal{G}$. Note that there is exactly one path from any internal node in the DAG to any leaf. Each node in D has a buffer. Thread-local nodes have a buffer of constant capacity, leaf nodes have buffers of capacity $O(\alpha \log^2 V)$ updates and other global nodes have buffers of capacity C updates. It is important that the leaf nodes have a limited capacity to ensure that the size of the pipeline hypertree does not grow beyond that of the CameoSketches. By this construction, each leaf has size $O(\alpha \log^3 V)$ bits (a constant factor α greater than that of a CameoSketch). Figure 14 illustrates the design.

The API for the pipeline hypertree supports two operations: insert(update) and force_flush(). During an insert, update $e = (u, v)$ a thread inserts (u, v) and (v, u) into the buffer of its root node. When any non-leaf buffer fills, each update (j, k) in the buffer is moved to the child on the path to leaf j. We call this operation a ***flush***. Moving elements to children may fill the buffer of a child and cause the child to flush as well. Finally, when a leaf buffer fills, its contents are packaged as a batch and sent over the network for update processing. The force_flush operation allows a user to force all buffered updates to be moved through the pipeline hypertree and into the CameoSketches. This is accomplished by performing a flush operation on each node of D in a breadth first search pattern.

The pipeline hypertree design avoids unnecessary cache misses and thread contention. We only accesses a node of the DAG D when we have many updates to move into the node. As a consequence, the amortized cost of placing a single update into the vertex-based batch is less than a single cache miss. This is why we do not simply use a hash table (which costs at least one cache miss per update) to perform the batching. Additionally, the top levels of D are kept thread-local to avoid synchronization. Keeping shared data lower in D reduces the likelihood that two threads touch the same node simultaneously.

It is immediate from the construction of the pipeline hypertree that each update is moved $O(\log_{C/\mathcal{L}} V)$ times and that the total size of the data structure is $O(V \log^3 V)$ bits. Thus, pipeline hypertree achieves Claim 1.4 and does not violate Claim 1.1. In Appendix E.2 we describe our implementation of the pipeline hypertree, and in Section 7.2 we show it is crucial for fast stream ingestion.

D Proofs of Theorems 5.2 and 5.3

Theorem D.1. (Communication cost.) *The communication cost of ingesting N updates and answering Q queries is at most $(3 + 1/(\gamma\alpha))N$, where γ and α are constants.*

Proof. In the worst case, the input stream sends $O(\alpha\gamma V\phi)$ edge updates such that each vertex $u \in \mathcal{V}$ is an endpoint of exactly $\alpha\gamma\phi$ of these edges. Then it issues a connectivity query. At this point, each leaf (which has size $\alpha\phi$ is exactly a γ-fraction full, and so sends the contents of each leaf as a vertex-based batch to a distributed worker. The worker responds with a sketch delta of size ϕ, so the average communication induced per edge update is $O(\alpha\gamma)$.

Note that no query can induce more communication that this - if any leaf buffer is less than a γ-fraction full, the updates it contains are processed locally and incur no communication. If any leaf buffer is more than a γ-fraction full, the average communication per update in that buffer is lower (since the sketch delta is always a fixed size).

If the input stream alternates sending $O(\alpha\gamma\phi V)$ edge updates as above and issuing connectivity queries, the average communication cost per stream update is $O(\alpha\gamma)$, regardless of stream length. ∎

Theorem D.2. (Computational cost on main node.) *Given a input stream of length N defining $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a series of Q connectivity queries issued throughout the stream, let N_i denote the number of edge updates that arrive after query Q_i but before query Q_{i+1}.* Landscape *never uses more than $O(V \log^3 V)$ bits of space on the main node while processing the stream, and the amortized cost per update to process N_i is $O(\log_{C/\mathcal{L}}(V))$ if $N_i = \Omega(V \log^2(V))$ and $O(\log(V))$ otherwise. Computing each query Q_i takes $O(V \log^2(V))$ time.*

Proof. The space bound is immediate from the size of $\mathcal{S}(\mathcal{G})$ and the pipeline hypertree.

Assuming $N_i = \omega(\alpha V \log^3 V)$, the average number of updates per graph vertex is $\omega(\alpha \log^3 V)$. If a constant fraction of all graph vertices receive fewer than $O(\alpha\gamma \log^2 V)$ updates (and therefore process these updates locally) the total CPU work done on the main node to process all updates is $O(\log V \cdot V\alpha\gamma \log^2 V + \log_{C/\mathcal{L}} V \cdot V \cdot \alpha \log^3 V)$ and therefore the average work per update is $\log_{C/\mathcal{L}}$. On the other hand, if $N_i = O(\alpha V \log^3 V)$ in the extreme case all batches may be processed locally and then the CPU work per update is $O(\log V)$. ∎

E Landscape Implementation

Landscape must handle two tasks: stream ingestion, where edge updates from the input stream are compressed into the graph sketch; and query processing, where connectivity queries are computed from the graph sketch (or sometimes from auxiliary query-accelerating data structures, described below). In this section we describe the implementation of StreamIngestor, which handles stream ingestion, and QueryProcessor, which handles answering queries.

Processing stream updates into the graph sketch is a computationally intensive process: for a moderately sized data-set with 2^{18} vertices, applying a single edge update

requires evaluating 184 hash functions. LANDSCAPE farms out this computationally intensive portion of the workload to worker nodes while the other portions of stream ingestion, including update buffering and sketch storage, remain the responsibility of the main node. LANDSCAPE uses $164V * (\log^2 V - \log V)$B of space on the main node to store the sketches and the pipeline hypertree. On the worker nodes, LANDSCAPE requires storage for a single sketch and batch per CPU. Thus, a worker node with t threads requires $t \cdot 164(\log^2 V - \log V)$ bytes. On a billion-vertex graph, each worker thread requires only 64 KiB of RAM. When computing k-connectivity, all of the above costs are multiplied by a factor k.

E.1 Distributed Communication We use OpenMPI for message passing over the network. The main node stores the graph sketch and the pipeline hypertree. As edge updates arrive from the input stream, the main node inserts them into the pipeline hypertree and performs flushes when necessary. When a leaf fills, the main node sends the contents of the leaf (a vertex-based batch) as an OpenMPI message to a worker node.

E.2 StreamIngestor Design LANDSCAPE's STREAMINGESTOR processes information from the input stream into the graph sketch. Stream updates go through three key steps on their way to being applied to the graph sketch: first the updates are collected into vertex-based batches in the pipeline hypertree on the main node, then these batches are processed into sketch deltas by the worker nodes, and finally each sketch delta is added to the graph sketch on the main node. A high level description of STREAMINGESTOR is outlined in Figure 2. We summarize the design of each component below.

Pipeline Hypertree.

We chose the parameters of the pipeline hypertree to balance insertion performance and memory footprint. Levels 0, 1, and 2 in the pipeline hypertree are thread-local. Each thread has a private copy of all thread-local nodes it owns. Each thread owns eight level-0 nodes. At level i for $i \in [0, 2]$ each node has a buffer of size $8 \cdot 2^i$KiB and a fan-out of $16 \cdot 2^i$. Levels 3 and 4 are global levels: any thread may read to or write from any node in these levels. Each level 3 node has a buffer of size $\frac{512 \cdot V}{8 \cdot 2^3}$KiB and a fan-out of $\frac{V}{8 \cdot 2^3}$. There are exactly V nodes in level 4 node and each has a buffer of size $\alpha \log^2 V$ (or $k\alpha \log^2 V$ for k-connectivity).

Updates are assigned to threads arbitrarily. Our implementation reads streams from a file in parallel; whichever thread reads the update is responsible for buffering it.

When a leaf buffer becomes full, its contents are placed into the Work Queue for processing by worker nodes.

Work Queue. LANDSCAPE's main node uses two types of threads to send vertex-based batches to worker nodes: Graph Insertion threads produce batches of updates from the pipeline hypertree, and Work Distributor threads send these batches over the network for distributed processing. Data flow between Graph Insertion and Work Distributor threads is synchronized by the ***Work Queue***, a many-producer, many-consumer queue for seamless communication between Graph Insertion threads (producers) and the Work Distributor threads (consumers).

The Work Queue uses two linked lists and two mutexes/condition variables to coordinate simultaneous operations. Threads interacting with the Work Queue only need to hold locks for a constant amount of time, allowing us to achieve a high degree of parallelism. This is because both Work Queue operations perform only pointer swapping within their critical regions to move vertex-based batches into and out of the queue.

Generating sketch deltas. To generate a sketch delta from a batch of updates for vertex u, the worker node applies each update to an initially empty vertex sketch $S_u(\varnothing)$. We summarize the process of updating CAMEOSKETCHES (see Fig. 12). A vertex sketch consists of $\log_{3/2}(V)$ CAMEOSKETCHES that are modified by each edge update. Each CAMEOSKETCH stores a $O(\log(V^2)) \times 2$ matrix of "buckets". Finally, each bucket is defined by two variables α and γ.

Performing a CAMEOSKETCH update requires performing $1 + (\log(1/\delta) = 2) = 3$ hash calls (we use xxHash [15]), one for each column and one to determine the checksum. Once we have computed the hash values, we complete the update with four bitwise XORs. The cost of updating a sketch is dominated by the hash calls. To update a sketch with one edge update requires $3 \times \log_{3/2} V$ hash calls, for a graph on 2^{18} vertices this is 92, and we must update two sketches per edge insertion for a total of 184 hash calls.

E.2.1 Extending to k-connectivity LANDSCAPE can apply updates to k-connectivity sketches using essentially the same method as described for connectivity above. The size of a vertex-based batch (and consequently the size of leaf node buffers in the pipeline hypertree) is increased by a factor k to $164 \cdot \alpha k V(\log^2(V) - \log(V))$ bytes. To process a batch, the distributed worker produces sketch deltas for all k connectivity sketches, concatenates them, and sends them back to the main node.

E.3 QueryProcessor Design LANDSCAPE users may issue global connectivity or batched reachability queries while the stream is being processed. Issuing a query provides an answer accurate to the graph constructed by the stream up to the point the query was issued. To provide this accuracy requires pausing stream ingestion while processing the query so that all the sketches are in the correct state.

When a query is issued, if LANDSCAPE has a valid

GreedyCC instance then it uses it to answer the query as described in Section E.4. We show in Section 7 that answering queries this way is extremely fast. If there is no valid GreedyCC when the query is issued, Landscape instead flushes all pending updates out of the pipeline hypertree, applies them to its graph sketch, and then runs Borůvka's algorithm to compute connectivity.

Flushing and applying updates. All pending updates must be applied to the graph sketch before the query can be computed. When a query is issued, if there is no valid GreedyCC, Graph Insertion threads immediately flush the pipeline hypertree, forcing all updates into its leaves. As described in Section 5.3, Landscape distributes the work of processing the updates in each leaf provided the leaf is full enough; nearly-empty leaves are instead processed locally on the main node.

We chose a 4% fullness threshold for this policy because it gave the best performance on our cluster; Landscape users can change this default threshold value if desired. By Theorem 5.2 this can in the worst case lead to a 25× increase in network communication. We demonstrate in Section 7 that the network communication blowup on real input streams is within this bound (and is typically much lower).

Borůvka's algorithm with sketches. Once all sketch deltas have been merged into the graph sketch, we run Borůvka's algorithm to find the spanning forest of the graph defined by the input stream. Borůvka's algorithm proceeds over the CameoSketches as described in Section 4.

E.4 GreedyCC: Reusing Prior Queries.

Landscape is capable of reusing results from prior queries to drastically reduce latency for future queries. In response to a connectivity query q, Landscape uses Borůvka's algorithm to produce a spanning forest of the graph. Landscape retains the information from this spanning forest in a data structure we call GreedyCC and uses it for future queries.

Data structure. GreedyCC consists of a union-find data structure encoding the connected components of the spanning forest, and a hash table containing the edges of the spanning forest. Both of these data structures are compact: they require $O(V)$ space. After the query q, when edge update $e = (u, v)$ arrives from the input stream, in addition to inserting e into the pipeline hypertree, Landscape also uses e to update GreedyCC: if e is an edge insertion and u and v are not in the same connected component, a Graph Insertion thread merges their components in GreedyCC's union find in $O(\mathcal{A}(V))$ time, where $\mathcal{A}$ is the inverse Ackerman function [70]. It also adds e to the hash table in $O(1)$ expected time.

Query acceleration. If a second query q' is issued later, Landscape can use GreedyCC rather than recompute graph connectivity from scratch, which would take $O(V \log^3(V))$ time. For global connectivity queries, Landscape returns GreedyCC's spanning forest in $O(V)$ time and for batched reachability queries it uses GreedyCC's union-find data structure to compute reachability for all of the m vertex pairs in $O(m\mathcal{A}(V))$ time. We demonstrate experimentally in Section 7.3 that answering queries using GreedyCC is several orders of magnitude faster than the sketch Borůvka algorithm.

GreedyCC validity. However, if one of the edges $e = (u, v)$ in GreedyCC's spanning forest is deleted after the query, GreedyCC does not retain enough information to determine whether or not vertices u and v are still connected and, if they are, to find a replacement edge. Recovering this information is only possible by running the sketch Borůvka algorithm. When a spanning forest edge is deleted in this way, we say that GreedyCC has become ***invalid*** meaning it is no longer useful for answering connectivity queries. While Landscape maintains a valid GreedyCC, as each edge update $e = (u, v)$ arrives if e is a deletion Landscape checks whether or not e is in GreedyCC's hash table. If it is, Landscape discards GreedyCC as invalid. In the worst case, an adversarial input stream would render GreedyCC invalid immediately after each query by deleting a key edge but this behavior is pathological and, as we show in Section 7.2, unlikely to occur in real data streams. For practical data streams it is more likely that GreedyCC remains valid for a while after a query, during which time subsequent queries can be answered very quickly.

GreedyCC implementation. GreedyCC consists of a union-find data structure encoding the connected components of the spanning forest, and a hash table containing the edges of the spanning forest. Both of these data structures are compact: they require $O(V)$ space. After the query q, when edge update $e = (u, v)$ arrives from the input stream, in addition to inserting e into the pipeline hypertree, Landscape also uses e to update GreedyCC: if e is an edge insertion and u and v are not in the same connected component, a Graph Insertion thread merges their components in GreedyCC's union find in $O(\mathcal{A}(V))$ time, where $\mathcal{A}$ is the inverse Ackerman function [70]. It also adds e to the hash table in $O(1)$ expected time.

If a second query q' is issued later, Landscape can use GreedyCC rather than recompute graph connectivity from scratch, which would take $O(V \log^3(V))$ time. For global connectivity queries, Landscape returns GreedyCC's hash table in $O(V)$ time and for batched reachability queries it uses GreedyCC's union-find data structure to compute reachability for all of the k vertex pairs in $O(k\mathcal{A}(V))$ time. We demonstrate experimentally in Section 7.3 that answering queries using GreedyCC is several orders of magnitude faster than the sketch Borůvka's algorithm.

However, if one of the edges $e = (u, v)$ in GreedyCC's spanning forest is deleted after the query, GreedyCC does not retain enough information to determine whether

or not vertices u and v are still connected and, if they are, to find a replacement edge. Recovering this information is only possible by running the sketch Borůvka's algorithm. When a spanning forest edge is deleted in this way, we say that GreedyCC has become ***invalid*** meaning it is no longer useful for answering connectivity queries. While Landscape maintains a valid GreedyCC, as each edge update $e = (u, v)$ arrives if e is a deletion Landscape checks whether or not e is in GreedyCC's hash table. If it is, Landscape discards GreedyCC as invalid. In the worst case, an adversarial input stream would render GreedyCC invalid immediately after each query by deleting a key edge but this behavior is pathological and, as we show in Section 7.2, unlikely to occur in real data streams. For practical data streams it is more likely that GreedyCC remains valid for a while after a query, during which time subsequent queries can be answered very quickly. In Section 7.3 we show that GreedyCC reduces query latency by up to four orders of magnitude.

F More Experiments

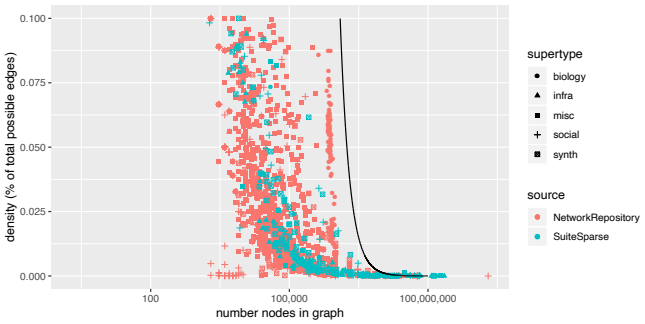
In this section we present several additional experiments, as well as additional detail about several experiments described in the main body of the paper.

F.1 Dense Graph Survey: Further Analysis In the introduction, we argue via Figure 1 that the lack of large, dense datasets in academic papers is likely a selection effect. The figure illustrates that graph datasets from a variety of applications and sourced from several collections rarely are larger than roughly 16GB.

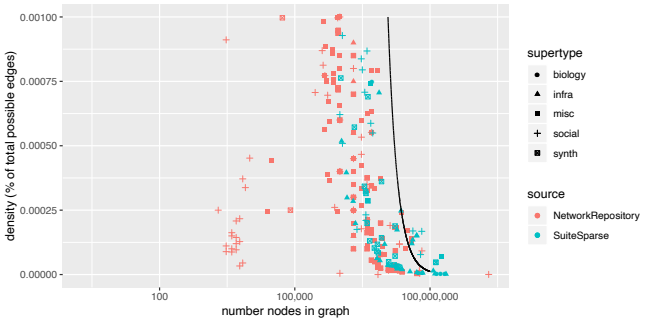
Figures 15a and 15b are alternative visualizations of the same collections of graph datasets. In Figure 15a we restrict the y axis to show graphs whose density is at most 0.1% of all possible edges, and in Figure 15b graphs of at most 0.001% density. The selection effect is even easier to see in the plots: as vertex count increases, density decreases such that the adjacency list size is not more than low double-digit gigabytes.

F.2 Correctness The connectivity sketch underlying Landscape is a randomized algorithm which succeeds with high probability - that is, the probability of it returning an incorrect spanning forest is bounded above by $1/V^c$ for some constant $c > 1$, which is a function of the sketch failure parameter δ. We set $\delta = 0.01$ and it is straightforward to show that consequently $c > 2$. As a result, the analytical failure probability of the connectivity algorithm for $V > 10,000$ is less than 10^{-8} regardless of the input stream.

To test correctness empirically, we compared the spanning forests output by Landscape with those computed from an adjacency matrix. We repeated this experiment 1000 times each for the kron17, p2p-gnutella, rec-amazon, google-plus, and web-uk streams. No failures were ever observed.



(a) Graphs of up to 0.1% maximum density.



(b) Graphs of up to 0.001% maximum density.

Figure 15: Alternative visualizations of the graph datasets from NetworkRepository and SuiteSparse, emphasizing the relationship between vertex count, maximum observed density and relatively small representation size (16GB).

F.3 Landscape k-connectivity performance Table 5 summarizes Landscape's performance for computing k-connectivity for a variety of datasets and values of k. All experiments were run with 32 distributed workers for a total of 512 threads.

F.4 Comparison to GraphZeppelin For completeness, we compare Landscape's ingestion rate and query latency against GraphZeppelin because it uses sketching and is opimized for dense graphs. The following experiments were run on a single AWS c6a.48xlarge instance (with 192 cores) for direct comparison since GraphZeppelin is a single-machine system.

We evaluated both systems' ingestion rates on the kron17 stream, varying the number of threads. Figure 16 summarizes the results. We see that even at a small number of threads Landscape is almost an order of magnitude faster than GraphZeppelin; this is a consequence of the reduced update time of CameoSketch. Further, GraphZeppelin fails to scale beyond 80 threads, with a

Table 5: LANDSCAPE performance for computing k-connectivity on a variety of real-world and synthetic datasets. N/A entries indicate the experiment was not run because the sketch was larger than available RAM.

	Insertions per second (millions)				Memory Consumption (GiB)			
Dataset	k = 1	k = 2	k = 4	k = 8	k = 1	k = 2	k = 4	k = 8
ca-citeseer	17.29	8.99	4.71	2.37	14.51	24.99	42.87	79.57
google-plus	108.6	65.4	37.8	2.06	9.68	13.62	21.99	34.80
p2p-gnutella	14.79	8.13	4.79	2.48	7.22	9.16	13.26	19.44
rec-amazon	13.55	7.64	4.48	2.28	8.68	12.59	19.51	30.11
web-uk	91.5	54.7	2.87	14.95	11.17	16.38	28.62	48.58
kron13	177.7	113.3	46.9	23.67	5.75	5.94	6.68	7.49
kron15	314.3	197.7	102.6	50.86	7.72	10.02	15.46	20.73
kron16	329.7	207.3	105.2	52.67	10.08	14.43	25.16	41.32
kron17	338.5	200	101.5	50.76	15.25	24.40	46.49	83.39
erdos18	309.7	189 94.7	94.7	N/A	26.04	45.90	88.90	N/A
erdos19	234.1	170.9	N/A	N/A	50.98	93.20	N/A	N/A
erdos20	245.8	N/A	N/A	N/A	105.93	N/A	N/A	N/A

	Query Latency (seconds)				Network Communication (GiB)			
Dataset	k = 1	k = 2	k = 4	k = 8	k = 1	k = 2	k = 4	k = 8
ca-citeseer	2.447	7.55	20.25	38.83	0.11	0.207	0.408	0.808
google-plus	1.03	3.664	11.54	38.51	4.91	7.798	13.34	24.74
p2p-gnutella	0.42	1.301	3.29	8.548	0	0	0	0
rec-amazon	0.87	2.323	4.296	8.22	0	0	0	0
web-uk	1.13	3.715	5.682	10.91	4.8	8.322	15.41	29.60
kron13	0.06	0.219	0.7647	2.866	1.72	1.93	2.061	2.287
kron15	0.284	1.076	3.372	13.95	26.8	29.5	30.22	31.68
kron16	0.581	1.95208	7.02079	25.6338	106.6	116.864	118.562	121.887
kron17	1.27	5.02412	16.1608	65.5716	425.2	464.981	468.886	476.598
erdos18	3.15	8.71687	41.5415	N/A	545	597.472	605.368	N/A
erdos19	6.97	18.9148	N/A	N/A	551.4	607.217	N/A	N/A
erdos20	14.1	N/A	N/A	N/A	1377	N/A	N/A	N/A

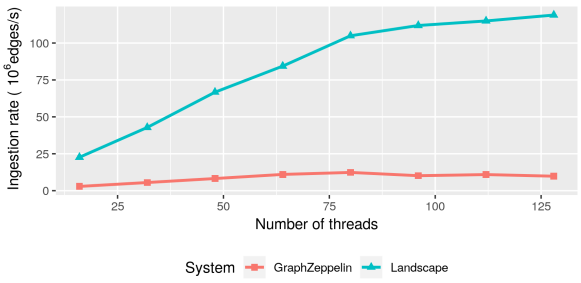


Figure 16: Single-machine scaling performance of GRAPHZEPPELIN and LANDSCAPE. GRAPHZEPPELIN's maximum ingestion rate is less than 15 million. LANDSCAPE avoids this bottleneck due to CAMEOSKETCH and the pipeline hypertree.

maximum ingestion rate of less that 15 million updates/sec. To explain this, we also compared the throughput of each systems' buffering systems in isolation on our cluster's main node. Since GRAPHZEPPELIN's in-RAM buffering data structure is not designed to minimize L2 cache misses, its performance is nearly two orders of magnitude slower than sequential RAM bandwidth (and therefore much slower than LANDSCAPE). In contrast, the throughput of LANDSCAPE's pipeline hypertree increases to over 500 million updates/sec.

We also repeated our query latency experiment on GRAPHZEPPELIN, which took roughly 1 second to compute each global connectivity or batched reachability query. While this is slightly lower latency on the first query in a burst than LANDSCAPE, LANDSCAPE's GREEDYCC heuristic results in four orders of magnitude lower latency on subsequent queries of both types.

We conclude that LANDSCAPE achieves greater performance than GRAPHZEPPELIN, because of CAMEOSKETCH, the pipeline hypertree, and GREEDYCC.

G Related Work.

Single-Machine Streaming Graph Systems. Many existing single-machine graph stream processing systems are

optimized for the ***batch-parallel*** model where stream updates arrive in large batches consisting entirely of insertions or entirely of deletions. Some batch-parallel systems answer queries synchronously, meaning they stop ingesting stream updates while computing queries [60, 11, 21, 55, 69, 68]. Others periodically take "snapshots" of the graph during ingestion that enable asynchronous query evaluation [18, 14, 35, 37, 51]. Some systems are designed to process only streams of edge insertions [63, 52].

Distributed Streaming Graph Systems. Kineograph [13] is a general-purpose framework for incremental graph algorithms which uses snapshots for efficient incremental computation. GraphTau [34] uses a graph snapshot method implemented on top of Apache Spark's RDDs for graph analytics in the sliding window model. Kickstarter [74] is a runtime technique for incremental monotonic graph algorithms which trims approximate vertex values when edges are deleted for efficient result updating. It is implemented in the distributed graph processing system ASPIRE[75]. While Kickstarter is a potentially interesting point of comparison, its source code is not publically available. CellIQ [35] is a distributed graph system optimized for solving the connected components problem on cellular network graphs in the sliding window model. Tegra [36] is designed to compute a variety of time-window graph analytics on dynamic graphs. DiStinger [26] is a distributed extension of Stinger [21] which is optimized for solving PageRank on sparse graph streams.

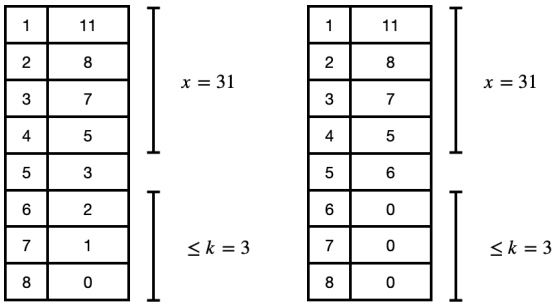
H Analysis of CameoSketch's Space Usage

This section proves Theorem 4.3. Our analysis discusses a single sketch column and then generalizes to a sketch with multiple columns. Since all buckets have the same column we drop the column from our notation and refer to the column 0, row r bucket as b_r.

We construct our sketches using a ***k-wise independent*** hash function. The size of a bucket $|b_i|$ is equal to the number of nonzero indices that hash to depth i. For CameoSketch this is also equivalent to the number of indices held by the bucket. $|b_i|$ is a random variable that depends upon the hash function and the set of nonzeros. Let $\boldsymbol{Z}$ be the set of nonzero indices in the input vector and let $\boldsymbol{z} = |Z|$. The maximum depth (number of rows) of the column is $d = O(\log n)$.

The ***k-boundary bucket*** (for a constant k) is the bucket of smallest depth for which $\leq k$ non-zero vector indices have depth greater than or equal to the boundary. That is the boundary bucket is $\min_i \sum_{j=i}^{d} |b_j| \leq k$.

Our analysis will assume that $z > k$ as otherwise we can directly apply the fully independent results that we will get to below.



(a) Maximum number of non-zeros below a 3-boundary bucket.

(b) Minimum number of non-zeros below a 3-boundary bucket.

Figure 17: Example of a k-boundary bucket (where $k = 3$) for a sketch column with bucket depth and the size of each bucket. In this example, bucket 6 is the boundary bucket and the total number of indices in the column is 36. The total number of vector indices with depth 1 through 4 is $x = 31$. Given that b_6 is the boundary bucket and $x = 31$ (the number of indices in b_4 or above), the number of vector indices with depth 5 is in range $[3, 6]$.

Lemma H.1. *If bucket $b_{\beta+1}$ is a k-boundary bucket, the sizes of the buckets $[\beta, d]$ are determined by the outcome of k random variables. Specifically, let $t = \sum_{j=1}^{d} |b_j|$ be the total number of vector indices that appear in the entire column (but not the deterministic bucket). If bucket $b_{\beta+1}$ is a k-boundary bucket and the number of indices that map to buckets above b is $x = \sum_{j=1}^{\beta-1} |b_j|$, then there exists a set $F \subseteq Z$ such that $|F| = k$, $|b_\beta \cap F| + t - x = S_\beta(Z)$, and $\forall i \in [\beta + 1, d]$, $|b_i \cap F| = |b_i|$.*

Proof. By construction, bucket b_β has size of at least $t - x - k$ and at most $t - x$. If this were not the case, then bucket $b_{\beta+1}$ could not be the boundary bucket. Assume for the sake of contradiction that this condition does not hold. Thus, $|b_\beta|$ must either by less than $t - x - k$ or greater than $t - x$. The case of $|b_\beta| > t - x$ is straightforwardly impossible as $x + |b_\beta|$ would be greater than t. In the case where $|b_\beta| < t - x - k$, then either $|b_\beta|$ is negative (an impossibility) if $x + k > t$ or $\sum_{j=\beta+1}^{d} |b_j| > k$ thus violating the assumption that $b_{\beta+1}$ is a k-boundary bucket.

Given that we know that $|b_\beta|$ is the sum of z indicator random variables and the the value of $|b_\beta|$ can only vary by at most k, there must exist a set of k indicator random variables whose outcome defines $|b_\beta|$. Additionally, since at most $k \leq t - x - |b_\beta| \geq 0$ non-zeros may appear in buckets $[\beta + 1, d]$ by construction, there must exist a set of k non-zeros whose depth determines the size of all buckets $[\beta, d]$. ∎

We construct a ***k-isolated column***: a sketch column that contains only the set F (that we defined in Lemma H.1) of size k non-zeros and the buckets in which

1	{T, U, V, W, X, Y, Z}
2	{B,C,H,J}
3	{A,E,F,G}
4	{D}
5	{}
6	{P}
7	{}

(a) Original column with 3-boundary bucket B_4. The indicated buckets will be isolated.

1	{A}
2	{D}
3	{}
4	{P}
5	{}

(b) A 3-isolated column. Notice that the first bucket contains only a single non-zero. However, in the original column this is a bad bucket. This is an example of why we always assume that B_1 in the isolated column is bad.

Figure 18: An example of creating an isolated column. Notice that the isolated column contains only three non-zeros bucket we preserve a mapping between good buckets in the isolated column and those in the original column.

they could appear. That is, we isolate a contiguous subset of the column's buckets and a subset of the non-zero indices. If the k-boundary bucket of a column C is $b_{\beta+1}$, then the k-isolated column $\hat{C}$ contains buckets $b_b, \ldots, b_d$.

Additionally, a k-isolated column is good if and only if a bucket of depth greater than one is good. That is, we assume bucket b_1 is always bad. An example of a column and the associated isolated column can be found in Figure 18.

LEMMA H.2. *Given bucket $\beta+1$ is the k-boundary bucket for the original column C, then the k-isolated column $\hat{C}$'s bucket $\hat{b}_i$ for $i \in [2, d-\beta+1]$ is good if and only if bucket $B_{i+\beta-1}$ is good in C.*

Proof. With the exception of the first bucket b_1 and the deterministic bucket b_0, the isolated column has buckets of identical content to those of the original column C. Thus, bucket $\hat{b}_i$ in the isolated column has identical content to bucket $b_{\beta+i-1}$ for $i \in [2, d-\beta+1]$, therefore, Bucket $\hat{b}_i$ is good if and only if bucket $b_{\beta+i-1}$ is good. ∎

We now prove that we can analyze the k-isolated column without needing to condition on the state of the original column. That is, we can view it as an entirely new column that only contains k nonzero elements.

LEMMA H.3. *Let column C_h have depth d and be constructed with a k-wise independent hash function h with image $[2^d]$, and let C_h's k-isolated column "begin" at an index β. Then the probability distribution of the size of each bucket in the k-isolated column function is identical to that of a column $C_{h'}$ with depth $d-\beta+1$.*

Proof. Let bucket $b_{\beta+1}$ be the k-boundary bucket for the original column C_h (with $h : [n] \to [2^d]$). The k-isolated column's hash function can thus defined as

$$\hat{h}(i) = \lfloor h(i)/2^{\beta-1} \rfloor$$

Note that, by the definition of k-wise independent hashing, for all $y \in [2^d]$, and for all $x \in [n]$

$$\Pr[h(x) = y] = \frac{1}{2^d}.$$

It follows directly from this that

$$\begin{aligned}\Pr[\hat{h}(x) = y] &= \Pr[\lfloor h(x)/2^{\beta-1} \rfloor = y] \\ &= \sum_{i=0}^{2^{b-1}-1} \Pr[h(x) = y \cdot 2^{\beta-1} + i] \\ &= \sum_{i=0}^{2^{b-1}-1} \frac{1}{2^d} = \frac{1}{2^{d-\beta+1}},\end{aligned}$$

for all x and y. Therefore, the distribution of a single hash call is uniform. Likewise, note that $\hat{h}$ is still k-wise independent since functions of k-wise independent variables are also k-wise independent. Thus, $\hat{h}$ satisfies the desired properties of a k-wise independent hash function, defined over $[n] \to [2^{d-\beta+1}]$.

Therefore, the sizes of the buckets constructed using $\hat{h}$ have an equivalent distribution to buckets constructed from hash function $h' : [n] \to [2^{d-\beta+1}]$ that is drawn independently (from the choice of h) from a family of k-wise independent hash functions. The probability distributions of these hash functions on the input to the k-isolated column are identical, that is, $\forall i \in [1, d-\beta+1]$, $P[\log(\hat{h}(x)) = i] = P[\log(h'(x)) = i] = P[\log(h(x)) = i+\beta-1 \mid \log(h(x)) \geq \beta]$. This result follows from the above statements about the distribution of $\hat{h}$. This statement holds for all inputs to the k-isolated column as by definition their depth must be at least β in the original column. ∎

Fully Independent Hash Functions The success probability of CAMEOSKETCH constructed with a fully independent hash function with $z > 1$ nonzeros (if 1 nonzero then the column succeeds deterministically) and column depth $d > 0$ is given by the function $F(z, d)$. Where F is defined by the following recurrence:

$$F(a,b) = \sum_{i \in [0,a] \setminus \{1\}} \left(2^{-a} \binom{a}{i} F(a-i, b-1) \right) + a2^{-a}$$

and if $a \leq 0$ or $b \leq 0$ then the probability of success is 0. For this recurrence, we look at the bucket at depth 1,

Non-zero Indices	CameoSketch
1	1
2	0.666
3	0.856
4	0.799
5	0.813
6	0.810
7	0.810

Table 6: Lower bound on the probability that a CameoSketch column (with 10 buckets) succeeds given a small number of non-zero indices, and with full independence.

and consider every possible number of items that can go in that first bucket and the probability of that scenario occurring. For each of these we calculate the likelihood that a bucket below is correct recursively in the case that the first bucket does not have exactly one element.

If we exclude the first bucket being good from our definition of success (which is required for our analysis of a k-isolated column), then the probability that the column succeeds is $\hat{F}(z,d) = F(z,d) - z2^{-z} \cdot (1 - F(z-1, d-1))$. Specifically, we are accounting for the possibility that the first bucket is good: $z2^{-z}$, and the rest of the buckets are bad: $1 - F(z-1, d-1)$.

For small values of z, we can solve this recurrence manually. Furthermore, note that though our hash functions are taken to have limited k-wise independence, as long as z is less than our choice of k, then analysis with these recurrences still holds.

Additionally, F is non-decreasing with respect to d, that is for any z, $F(z,d_1) \geq F(z,d_2)$ as long as $d_1 \geq d_2$. Since $\forall z, d$, $F(z,d) \geq 0$, it must be the case that adding additional subproblems (by increasing d) without changing the existing subproblems can only maintain or increase the likelihood of success.

Extending to k-wise independence Now we prove our theorems that analyze sketches constructed with k-wise independent hash functions. Our key strategy is to use k-isolated columns and the associated lemmas to reduce to a column on only k non-zero elements. Thus, we can perform fully independent analysis on this subproblem to solve for the probability the entire column succeeds.

Lemma H.4. *With $k \geq 3$-wise independent hash functions* CameoSketch *sketch columns have a success probability of at least* 0.66.

Proof. If the number of non-zeros is at most 3, we can solve for the probability of success assuming full independence. Table 6 gives the likelihood of success for 1, 2 and 3 non-zeros which are all greater than 0.66.

Otherwise, we analyze a 3-isolated column to lower bound the success probability of a sketch column C_h constructed from a 3-wise independent hash function h when $z > 3$. We will assume $n \geq 2^{10}$ (and artificially increase the size of the vector and thus sketch if it is not). Additionally, let the depth of the column be $d = \log n + 5$.

By Lemma H.1 and Lemma H.2, we can analyze 3 non-zeros to lower bound the success probability of a sketch on $z > 3$ non-zeros. Then, by Lemma H.3 we can perform this analysis assuming these 3 non-zeros exist within an isolated column of depth $d - b + 1$. This 3-isolated column has a depth function with probability distribution equivalent to that of a regular column with depth $d - \beta + 1$.

Solving $\hat{F}$ for $z = 3$ (i.e., with the additional assumption that the first bucket is always bad) gives a success probability of greater than 0.69 for CameoSketch if $d \geq 5$.

We also need to account for the probability that the k-boundary bucket is too far down in the sketch column, that is $d - \beta + 1$ is too small to give a high likelihood of success. To do so, we assume that if β is of too high depth, then the probability of success is zero. If β is of depth less than or equal to $\log n + 1$, then the analysis holds.

The probability that the boundary is greater than this cutoff is $1 - \binom{z}{3} 2^{-3(\log n + 2)}$. Recall that $n \geq 2^{10}$, at this vector size the likelihood that the boundary occurs at the depth $\log n + 2$ bucket or less is 0.98 and so the overall probability of success is $0.69 \cdot 0.98 > 2/3$.

Therefore, CameoSketch column on a vector of length n with $\log n + 5$ buckets (to assure that $d - \beta + 1$ is at least 5) has a success probability of at least 2/3. ∎

This improved probability of success per column means that our sketches can be much more compact while retaining the probability guarantees we require. Thus, we can finally prove Theorem 4.3.

Recall Theorem 4.3 *Using 3-wise independent hash functions,* CameoSketch *requires* $8 \log_3(1/\delta)(\log n + 5)$ *bytes of space to return a nonzero element of a length* $n < 2^{64}$ *input vector w/p at least* $1 - \delta$.

Proof. With 3-wise independence each CameoSketch column succeeds with probability at least 0.66 by Lemma H.4. Thus, the probability that a single column fails is 0.33. So to achieve a failure probability of δ requires $\log_{1/0.33}(1/\delta) \leq \log_3(1/\delta)$ columns. ∎

We do not know if CameoSketch samples nonzero elements uniformly. For the purposes of this paper it makes no difference as these connectivity algorithms require only some element from the set. They do not require a uniformly random element.

We can make the following simple modification to CameoSketch to ensure it samples uniformly. We

only count a bucket as good if it contains one nonzero and is the deepest nonempty bucket. Thus, we simulate CubeSketch's query procedure but retain the improved update time. Our improved analysis of the space constants also holds but requires more detail that we exclude here.