GraphZeppelin: How to Find Connected Components (Even When Graphs Are Dense, Dynamic, and Massive)

DAVID TENCH, Rutgers University, New Brunswick, USA
EVAN WEST, Stony Brook University, Stony Brook, USA
VICTOR ZHANG, Rutgers University, New Brunswick, USA
MICHAEL A. BENDER, Stony Brook University, Stony Brook, USA
ABIYAZ CHOWDHURY, Stony Brook University, Stony Brook, USA
DANIEL DELAYO, Stony Brook University, Stony Brook, USA
J. AHMED DELLAS, Rutgers University, New Brunswick, USA
MARTÍN FARACH-COLTON, Rutgers University, New Brunswick, USA
TYLER SEIP, MongoDB, New York, USA
KENNY ZHANG, Stony Brook University, Stony Brook, USA

Finding the connected components of a graph is a fundamental problem with uses throughout computer science and engineering. The task of computing connected components becomes more difficult when graphs are very large, or when they are dynamic, meaning the edge set changes over time subject to a stream of edge insertions and deletions. A natural approach to computing the connected components problem on a large, dynamic graph stream is to buy enough RAM to store the entire graph. However, the requirement that the graph fit in RAM is an inherent limitation of this approach and is prohibitive for very large graphs. Thus, there is an unmet need for systems that can process dense dynamic graphs, especially when those graphs are larger than available RAM.

We present a new high-performance streaming graph-processing system for computing the connected components of a graph. This system, which we call GraphZeppelin, uses new linear sketching data structures (CubeSketch) to solve the streaming connected components problem and as a result requires space asymptotically smaller than the space required for a lossless representation of the graph. GraphZeppelin is optimized for massive dense graphs: GraphZeppelin can process millions of edge updates (both insertions and deletions) per second, even when the underlying graph is far too large to fit in available RAM. As a result GraphZeppelin vastly increases the scale of graphs that can be processed.

CCS Concepts: • **Theory of computation → Sketching and sampling**; • **Mathematics of computing → Graph algorithms;**

Additional Key Words and Phrases: Linear sketching, streaming algorithms, external memory

Authors' addresses: D. Tench, V. Zhang, J. A. Dellas, and M. Farach-Colton, Rutgers University 57 US Highway 1, New Brunswick, NJ 08901, United States; e-mails: dtench@pm.me, victor@vczhang.com, jad525@scarletmail.rutgers.edu, martin@farach-colton.com; E. West, M. A. Bender, A. Chowdhury, D. Delayo, and K. Zhang, Stony Brook University 100 Nicolls Road, Stony Brook, NY 11794, United States; e-mails: etwest@cs.stonybrook.edu, bender@cs.stonybrook.edu, abchowdhury@cs.stonybrook.edu, ddelayo@cs.stonybrook.edu, kzzhang@cs.stonybrook.edu; T. Seip, MongoDB 1633 Broadway, FL 38, New York, NY 10019, United States; e-mail: tylerjseip@gmail.com.

ACM 0362-5915/2024/05-ART9
https://doi.org/10.1145/3643846

ACM Reference Format:
David Tench, Evan West, Victor Zhang, Michael A. Bender, Abiyaz Chowdhury, Daniel Delayo, J. Ahmed Dellas, Martín Farach-Colton, Tyler Seip, and Kenny Zhang. 2024. GraphZeppelin: How to Find Connected Components (Even When Graphs Are Dense, Dynamic, and Massive). *ACM Trans. Datab. Syst.* 49, 3, Article 9 (May 2024), 31 pages. https://doi.org/10.1145/3643846

1 INTRODUCTION

Finding the connected components of a graph is a fundamental problem with uses throughout computer science and engineering. A recent survey by Sahu et al. [72] of industrial uses of algorithms reports that, for both practitioners and academic researchers, connected components were the most frequently performed computation from a list of 13 fundamental graph problems that include shortest paths, triangle counting, and minimum spanning trees. It has applications in scientific computing [68, 76], flow simulation [77], metagenome assembly [28, 64], identifying protein families [58, 82], analyzing cell networks [5], pattern recognition [32, 40], graph partitioning [50, 51], random walks [38], social network community detection [46], graph compression [39, 49], medical imaging [34], and object recognition [33]. It is a starting point for strictly harder problems such as edge/vertex connectivity, shortest paths, and k-cores. It is used as a subroutine for pathfinding algorithms such as Djikstra and A^*, some minimum spanning tree algorithms, and for many approaches to clustering [25, 26, 67, 81].

The task of computing connected components becomes more difficult when graphs are very large, or when they are ***dynamic***, meaning the edge set changes over time subject to a stream of edge insertions and deletions. Applications on large graphs include metagenome assembly tasks that may include hundreds of millions of genes with complex relations [28], and large-scale clustering, which is a common machine learning challenge [26]. Applications using dynamic graphs include identifying objects from a video feed rather than a static image [41] or tracking communities in social networks that change as users add or delete friends [10, 12]. And of course, graphs can be both large and dynamic. Indeed, Sahu et al.'s [72] survey reports that a majority of industry respondents work with large graphs (> 1 million nodes or > 1 billion edges) and a majority work with graphs that change over time.

A natural approach to computing the connected components on a large, dynamic graph stream is to buy enough RAM to store the entire graph. Indeed, dynamic graph stream processing systems such as Aspen and Terrace [23, 66] can efficiently query the connected components of a large graph subject to a stream of edge insertions and deletions when the graph fits in RAM. However, the requirement that the graph fit in RAM is prohibitive for most large graphs: for example, a graph with ten million nodes and an average degree of 1 million, using 2 B to encode an edge, would require 10 TB of memory. We show in Section 6 that the Aspen and Terrace graph representations are significantly larger than this lower bound.

In public graph-data-set repositories, most graphs are smaller than typical single-machine RAM sizes. As Figure 1 illustrates, nearly all graphs in Network Repository [70] can be stored as an adjacency list in less than 16 GB. This fixed memory budget furthermore implies that graphs with large numbers of vertices must be sparse. Similarly, the Stanford SNAP graph repository and the SuiteSparse repository have few graphs larger than 16 GB, and graphs with many nodes are always extremely sparse.

Large, dense graphs, we argue, are absent from graph repositories not because they are unworthy of study, but because there are few tools to analyze them. To illustrate: dense graphs do

A preliminary version of this article appeared in Tench et al. [75].

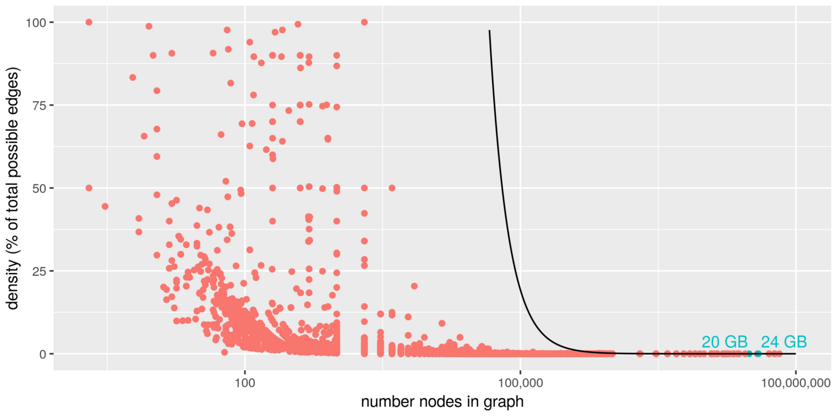


Fig. 1. Published graphs have few nodes or are sparse. Each point represents a graph dataset from NetworkRepository. Any point below the dark line indicates a graph that can be represented as an adjacency list in 16 GB of RAM.

appear in Network Repository [70], but these graphs are never larger than a few GB; moreover, as the graphs' vertex count increases, the maximum density decreases such that the densest graphs never require more than 10 GB. A compelling explanation for the absence of large, dense graphs is selection bias: interesting dense graphs exist at all scales, but large, dense graphs are discarded as computationally infeasible and consequently are rarely published or analyzed. Moreover, some large dense graphs are known to exist as proprietary datasets: for instance, 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. [19]

Thus, there is an unmet need for systems that can process dense graphs, especially when those graphs are larger than available RAM. Existing systems are not designed for large, dense, dynamic graph streams and instead optimize for other use cases. Aspen and Terrace are optimized for large, sparse, dynamic graphs that completely fit in RAM, and their performance degrades significantly on dense graphs and graphs larger than RAM. There is a deep literature on parallel systems for connected components computation in multicore [29], GPU [6], and distributed settings [14, 44] but these focus on static graphs which fit in RAM. Many external memory [13] and semi-external memory [1] systems focus on graphs that are too large for RAM and must be stored on disk, but none of these systems focus on graphs whose edges can be deleted dynamically.

In this article, we explore the general problem of connected components on large, dense, dynamic graphs. We introduce GraphZeppelin, which computes the connected components of graph streams using a $O(V/\log^3(V))$-factor less space than an explicit representation of the graph. GraphZeppelin uses a new ℓ_0-sketching data structure that outperforms the state-of-the-art on graph sketching workloads. Additionally, GraphZeppelin employs node-based buffering strategies that improve I/O efficiency. These techniques allow GraphZeppelin to scale better than existing systems in several settings. First, for in-RAM computation, GraphZeppelin's small size means it can process larger, denser graphs than Aspen or Terrace: specifically, dense graphs are twice as large as Aspen and at least 40 times larger than Terrace given 64 GB of RAM. Moreover, even if the input graph fits in RAM on all systems, GraphZeppelin is up to 3.5 times faster than Aspen and 6 times faster than Terrace on large dense graphs. GraphZeppelin also has comparable query latency to Aspen and Terrace for sufficiently large or dense graphs. Finally, GraphZeppelin scales to SSD at the cost of a 29% decrease to ingestion rate, and is more than two orders of magnitude faster than Aspen and Terrace, which suffer significant performance degradation when scaling out of RAM.

GraphZeppelin employs a new sketch algorithm, overcoming a computational bottleneck of existing linear sketching techniques in the semi-streaming graph algorithms literature [21]. The asymptotically best existing streaming connected components algorithm is Ahn et al.'s StreamingCC [3, 62], which has asymptotically low space and update time complexity. StreamingCC relies on ℓ_0-sampling, which it uses to sample edges across arbitrary graph cuts. However, the best known ℓ_0-sampling algorithm suffers from high constant and polylogarithmic factors in its space and update time, as we show in Section 3. This overhead makes any implementation of the StreamingCC data structure infeasibly slow and large. GraphZeppelin employs what we call CubeSketch, a specialized ℓ_0-sampling algorithm for sampling edges across graph cuts, to solve the connected components problem. For large graphs, CubeSketch uses four times less space than the best general ℓ_0-sampling algorithm and can process updates more than three orders of magnitude faster.

GraphZeppelin also uses new write-optimized data structures to overcome prohibitive resource requirements of existing semi-streaming algorithms. Streaming algorithms have had a significant impact in large part because they require a small (polylogarithmic) amount of RAM. In contrast, graph semi-streaming algorithms have higher RAM requirements: for most problems on a graph with V nodes, sublinear RAM is insufficient to even represent a solution so $O(V\text{polylog}(V))$ RAM is typically assumed. With the large polylog factors, this is often more RAM than is feasible in practice; see Section 2. We propose the hybrid streaming model, which enjoys the memory advantage of the streaming model while allowing enough space in external memory to compute on dynamic graph streams. In this model, there is still $O(V\text{polylog}(V))$ space available, but only $O(\text{polylog}(V))$ of this space is RAM and the rest is disk, which may only be accessed in $O(\text{polylog}(V))$-size blocks. The simultaneous challenges in this model are to design algorithms that use small total space but also have low I/O complexity. While existing graph semi-streaming algorithms use small space, their heavy reliance on hashing and random access patterns make them slow on disk. We show that GraphZeppelin is simultaneously a space-optimal in-RAM semi-streaming algorithm and an I/O efficient external memory algorithm for the connected components problem. We also validate its performance experimentally, showing that GraphZeppelin can operate on modern consumer solid-state disk, increasing the scale of dynamic graph streams that it can process while incurring only a 29% cost to stream ingestion rate.

Results. In this article, we establish the following:

- **GraphZeppelin:** We present a new high-performance streaming graph-processing system for computing the connected components of a graph. This system, which we call GraphZeppelin, uses new linear sketching data structures (CubeSketch, described below) to solve the streaming connected components problem using only $O(V\log^3(V))$ bits—a $O(V/\log^3(V))$-factor less space than any lossless representation of the graph. GraphZeppelin is optimized for massive dense graphs: GraphZeppelin can process millions of edge updates (both insertions and deletions) per second, even when the underlying graph is far too large to fit in available RAM. As a result GraphZeppelin vastly increases the scale of graphs that can be processed.
- **CubeSketch: ℓ_0-sampling optimized for graph connectivity sketching.** We give a new ℓ_0-sampling algorithm, CubeSketch, for vectors of integers mod 2. Given a vector of length n and failure probability δ, CubeSketch uses $O(\log^2(n)\log(1/\delta))$ bits of space and $O(\log(n)\log(1/\delta))$ average time per update, which is a factor of $O(\log(n))$ faster than the best existing ℓ_0-sampler for general vectors [21].

CubeSketch is a key subroutine in GraphZeppelin, where it is used to sample graph edges across arbitrary cuts as part of connected components computation. Here it is used to sketch vectors of length $\binom{V}{2} = O(V^2)$, where V denotes the number of nodes in the graph. We show experimentally that CubeSketch's ingestion is more than 3 orders of magnitude faster than the state-of-the-art ℓ_0 sampling algorithm on graph streaming workloads, and its queries are two orders of magnitude faster.
In addition to the $O(\log(V))$-factor speedup, several non-asymptotic factors contribute to this performance improvement as well. First, the existing algorithm's average update cost is dominated by $O(\log(V)\log(1/\delta))$ division operations, while CubeSketch's average update cost is dominated by $O(\log(1/\delta))$ bitwise XOR operations, which are much faster. In addition, the general algorithm performs 128-bit arithmetic operations (including division) when processing graphs with more than 10^5 nodes, whereas CubeSketch can use standard 64-bit operations to achieve the same error probability. Finally, both algorithms match the asymptotic space lower bound but CubeSketch uses roughly 4 times less space than the general algorithm.

— **Asymptotic guarantees of GraphZeppelin: space-optimality, I/O efficiency, $O(\log(V))$ average time per update.** GraphZeppelin's core algorithm matches the $O(V\log^3(V))$-bit space lower bound for the streaming connected components problem, and its average per-update time cost of $O(\log(V))$ is $O(\log(V)))$ times faster than the best existing algorithm [3]. Additionally, GraphZeppelin can efficiently ingest stream updates even when its sketch data structure is too large to fit in RAM: its I/O complexity is $sort$(length of stream) + $O(V/B\log^3(V) + V\log^*(V))$ and for realistic block sizes it is an I/O-optimal external-memory algorithm [18]. As a result, given a fixed amount of RAM and disk, GraphZeppelin is capable of efficiently computing the connected components of larger graphs than existing algorithms in the streaming or external memory models.

— **Empirical achievements of GraphZeppelin: better scaling for in-memory, out-of-core, and parallel computation, and undetectable failure probability.** GraphZeppelin's CubeSketch-based design increases the size of input graphs that can be processed, scales well to persistent memory, and facilitates parallelism in stream ingestion. As a result, GraphZeppelin can ingest 2–5 million edge updates per second on a single scientific workstation (see Section 6), both when its data structures reside completely in RAM and also when they reside on fast disk. As a result of these advantages, GraphZeppelin is faster and more scalable than the state-of-the-art on large, dense graphs:
  - **GraphZeppelin handles larger graphs for in-RAM computation.** GraphZeppelin's space-efficient CubeSketch allows it to process graph streams larger than can be stored explicitly in a fixed amount of RAM and give it an asymptotic $O(V/\log^3(V))$ space advantage over state-of-art systems on dense graphs. Given the polylogarithmic factors and constants, we need to determine the actual crossover point where GraphZeppelin processes graphs more compactly than Aspen and Terrace. We show empirically that this crossover point occurs when the space budget is between 32 and 64 gigabytes. That is, for dense graphs on several hundred thousand nodes, GraphZeppelin is 40% more compact than Aspen and several times more compact than Terrace, and this advantage only increases for larger space budgets or input sizes. Additionally, for dense graph streams on 2^{18} nodes GraphZeppelin ingests updates six times faster than Terrace and three times as fast as Aspen.
  - **GraphZeppelin can use persistent memory to handle even larger graphs.** GraphZeppelin's node-based work buffering strategy facilitates out-of-core computation,

allowing GraphZeppelin to use SSD to increase the scale of graph streams it can process while incurring a small cost to performance. We show experimentally that GraphZeppelin ingests updates more than two orders of magnitude faster than Aspen and Terrace when all systems swap to disk, and that using SSD slows GraphZeppelin stream ingestion by only 29%.

– **GraphZeppelin's stream ingestion is highly parallel.** GraphZeppelin employs a node-based work buffering strategy that facilitates parallelism and improves data locality. We show experimentally that GraphZeppelin's multithreaded stream ingestion system scales well with more threads: its ingestion rate is 25 times higher with 46 threads than an optimized single-thread implementation.

2 PRELIMINARIES

2.1 Graph Streaming and Hybrid Graph Streaming

In the ***graph semi-streaming*** model [27, 61] (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 $O(V\text{polylog}(V))$ 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 Section 4, we additionally use a new variant of the graph semi-streaming model, which we call the ***hybrid graph streaming model*** (since it incorporates some components of the external memory model [78] into the semi-streaming model). In this model, there is an additional constraint on the type of memory available for computation: only $M = \Omega(\text{polylog}(V)) = o(V)$ RAM is available, and $D = O(V\text{polylog}(V))$ disk space is available. A word in RAM is accessed at unit cost, and disk is accessed in blocks of $B = o(M)$ words at a cost of B per access. Any semi-streaming algorithm can be run with this additional constraint, but may become much slower if the algorithm makes many random accesses to disk. The algorithmic challenge in the hybrid graph streaming model is to minimize time complexity (of ingesting stream updates and returning solutions) in addition to satisfying the typical limited-space requirement of the data stream model. In Section 4, we show how GraphZeppelin can be adapted to this model, and is both a space-optimal single pass streaming algorithm with $O(\log^2(V))$ update time and an I/O efficient external memory algorithm.

We now summarize the streaming connected components problem studied in [3]:

Problem 1 (The Streaming Connected Components Problem). *Given an insert/delete edge stream of length N that defines a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, return the sets of vertices that define the connected components of $\mathcal{G}$.*

In Section 4, we present an improved algorithm for the above problem and analyze its performance in the hybrid graph streaming model. The algorithms for the connected components problem that we study in this article are probabilistic and guarantee that the connected components they return are exactly correct with high probability.

The above models and problem definition assume that there is a single connected components query, which is issued at the end of the entire stream of edge updates. Dynamic graph processing systems ideally support answering queries interspersed with graph updates. In Section 5, we present a streaming graph processing system which is based on our improved algorithm but supports arbitrarily interspersed queries and edge updates. In this article, we assume a non-adaptive adversary generating the input stream, that is, edge updates cannot be a function of the answers to prior queries.

2.2 Prior Work in Streaming Connected Components

We summarize StreamingCC, Ahn et al.'s [3] semi-streaming algorithm for computing a spanning forest (and therefore the connected components) of a graph.

For each node v_i in G, define the ***characteristic vector*** f_i of v_i to be a 1-dimensional vector indexed by the set of possible edges in $\mathcal{G}$. $f_i[(j,k)]$ is only nonzero when $i = j$ or $i = k$ and edge $(j,k) \in \mathcal{E}$. That is, $f_i \in \{-1, 0, 1\}^{\binom{V}{2}}$ s.t. for all $0 \leq j < k < \binom{V}{2}$:

$$f_i[(j,k)] = \left\{ \begin{array}{ll} 1 & i = j \text{ and } (v_j, v_k) \in \mathcal{E} \\ -1 & i = k \text{ and } (v_j, v_k) \in \mathcal{E} \\ 0 & \text{otherwise} \end{array} \right\}$$

Crucially, for any $S \subset \mathcal{V}$, the sum of the characteristic vectors of the nodes in S is a direct encoding of the edges across the cut $(S, \mathcal{V} \setminus S)$. That is, let $x = \sum_{v \in S} f_v$ and then $|x[(j,k)]| = 1$ iff $(j,k) \in E(S, \mathcal{V} \setminus S)$.

Using these vectors, we immediately have a (very inefficient) algorithm for computing the connected components from a stream: Initialize $f_i = \{0\}^{\binom{V}{2}}$ for all i. For each stream update $s = ((u,v), \Delta)$, set $f_u[u,v] += \Delta$ and $f_v[u,v] += -\Delta$.

After the stream, run Boruvka's algorithm [63] for finding a spanning forest as follows. For the first round of the algorithm, from each a_i arbitrarily choose one nonzero entry (w,y) (an edge in $\mathcal{E}$ s.t. w = i or y = i). Add e_i to the spanning forest. For each connected component C in the spanning forest, compute the characteristic vector of C: $a_C = \sum_{v \in C} f_v$. Proceed similarly for the remaining rounds of Boruvka's algorithm: in each round, choose one nonzero entry from the characteristic vector of each connected component and add the corresponding edges to the spanning forest. Sum the characteristic vectors of the component nodes of the connected components in the spanning forest, and continue until no new merges are possible. This will take at most $O(\log(V))$ rounds.

The key idea to make this a small-space algorithm is to use "ℓ_0-sampling" [21] to run this version of Boruvka's algorithm by compressing each characteristic vector f_i into a data structure of size $O(\log^2(V))$ that can return a nonzero entry of f_i with constant probability.

Definition 1. A sketch algorithm is a δ ℓ_0-sampler if it is

(1) **Sampleable:** it can take as its input a stream of updates to the coordinates of a non-zero vector a, and output a non-zero coordinate $(j, f[j])$ of f. $\mathcal{S}(f)$ denotes the sketch of vector f.
(2) **Linear:** for any vectors f and g, $\mathcal{S}(f) + \mathcal{S}(g) = \mathcal{S}(f + g)$ and this operation preserves sampleability, i.e., $\mathcal{S}(f+g)$ can output a nonzero coordinate of vector $f + g$.
(3) **Low Failure Probability:** The algorithm may fail by either not returning an answer (a ***null*** answer) or returning an incorrect value for a coordinate of f (an ***incorrect*** answer). The algorithm may give a null answer with probability at most δ. The algorithm may give an incorrect answer with probability at most $1/V^c$ for some constant c.

For all ℓ_0-samplers in this article, $\mathcal{S}(f)$ is a vector and adding two sketches is equivalent to adding their vectors elementwise.

Lemma 1 (Adapted from [21], Theorem 1). *Given a 2-wise independent hash family $\mathcal{F}$ and an input vector of length n, there is an δ ℓ_0-sampler using $O(\log^2(n)\log(1/\delta))$ bits of space.*

We denote a ℓ_0 sketch of a vector x as $\mathcal{S}(x)$. Since the sketch is linear, $\mathcal{S}(x)+\mathcal{S}(y) = \mathcal{S}(x+y)$ for any vectors x and y. This allows us to process stream updates as follows: we maintain a running sum of the sketches of each stream update, which is equivalent to a sketch of the vector defined by the stream. That is, let a_i^t denote a_i after stream prefix S_t. For the jth stream update $s_j = ((i,x),\Delta)$ we obtain $\mathcal{S}(f_i^j) = \mathcal{S}(s_j) + \mathcal{S}(f_i^{j-1})$.

Linearity also allows us to emulate the merging step of Boruvka's algorithm by summing the sketches of all nodes in each connected component. We require an independent ℓ_0-sampler for each $v \in \mathcal{V}$ and each of the $O(\log(V)$ rounds of Boruvka. For each of these ℓ_0-samplers we set $\delta = 1/3$, so each ℓ_0-sampler is $O(\log^2(V)$ bits (From Lemma 1). We refer to the $O(\log V)$ ℓ_0-sampler data structures for a single $v \in \mathcal{V}$ as a ***node sketch***. As there are $O(V)$ node sketches, the total size of the entire data structure is $O(V \log^3(V))$. Recent work [62] has shown that this is asymptotically optimal.

The above description assumes that the exact number of nodes V is known *a priori*. This is not strictly necessary: All we need is a loose upper bound on the number of nodes we will eventually see. Given an upper bound U s.t. $V \leq U \leq V^c$ for some constant c, we can simply define f_i to have length $\binom{U}{2}$. The node sketch of f_i then has size $O(\log^3(U^2)) = O(\log^3(V))$. We create a node sketch for v_i the first time it appears in a stream update (v_i, v_j) so the total space cost is still $O(V \log^3(V))$. Similarly, even if nodes are identified in the input stream as arbitrary strings instead of integer IDs in the range $[V]$, we can use a hash function with range $[O(U^2)]$ to ensure that every node gets a unique integer ID with high probability.

3 ℓ_0-SAMPLING REVISITED

Existing ℓ_0-sampling algorithms are asymptotically small and fast to update, but in practice high constant and logarithmic overheads in size and update time prevent these algorithms from being useful for a streaming connected components algorithm. We now review some details of the best known ℓ_0-sampling algorithm and demonstrate experimentally that using it to emulate Boruvka's algorithm for graph connectivity would be prohibitively slow and would require an enormous amount of space. Then we introduce an ℓ_0-sketching algorithm which exploits the structure of the connected components problem to improve performance, and experimentally demonstrate that it is 4 times smaller and 3 orders of magnitude faster to update than the state-of-the-art.

The best known ℓ_0-sampling algorithm [21] is summarized in Figure 3. Given a vector $f \in \mathbb{Z}^n$, the data structure consists of a matrix of $\log(n)$ by $q\log(1/\delta)$ "buckets" (for some small constant q). Each bucket represents the values at a random subset of positions of f. This representation is lossy: we can recover a nonzero element of f from bucket $\mathcal{B}_{i,j}$ only when a single position in $\mathcal{B}_{i,j}$ is nonzero. Equivalently, the support of $\mathcal{B}_{i,j}$, denoted by $supp(\mathcal{B}_{i,j})$, is 1. If $supp(\mathcal{B}_{i,j}) = 1$, we say that $\mathcal{B}_{i,j}$ is ***good***, and say that it is ***bad***, otherwise. With probability $1-\delta$, $\exists i,j$ s.t. $\mathcal{B}_{i,j}$ is good and therefore we can recover a nonzero value from f. Each bucket includes a ***checksum*** that indicates whether it is good with high probability.

Each bucket $\mathcal{B}_{i,j}$ contains three values: $a_{i,j}$, $b_{i,j}$, and $c_{i,j}$. If $\mathcal{B}_{i,j}$ is good, then the checksum test on line 15 passes and $f[b_{i,j}] = a_{i,j}/b_{i,j}$. If the checksum test fails $\mathcal{B}_{i,j}$ is bad.

When a stream update (e,Δ) arrives, its membership in each bucket is determined using the hash function on line 3: if $hash(e) \equiv 0 \pmod{2^i}$ then e is in $\mathcal{B}_{i,j}$. If it is in bucket $\mathcal{B}_{i,j}$, it is applied to $a_{i,j}$, $b_{i,j}$, and $c_{i,j}$ according to the logic on lines 7–9. When the sketch is queried, it checks whether each bucket passes the checksum test on line 15. If some bucket passes this test, its sampled value is returned. Figure 2 gives an example of this process. For a more thorough analysis of this algorithm see [21].

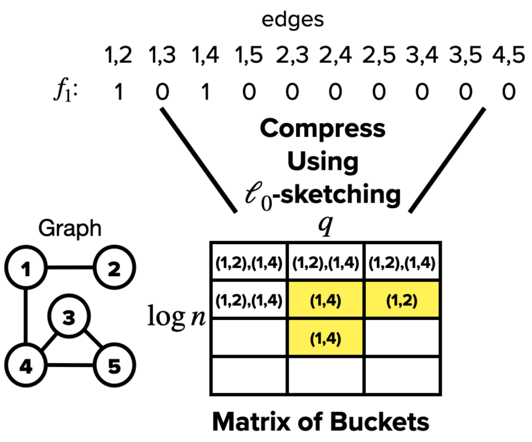


Fig. 2. Compressing a characteristic vector. Each highlighted cell contains one nonzero element from the vector and can be sampled, yielding an edge incident to node 1.

```
1:  function UPDATE_SKETCH(idx, Δ)                                  ▷ Add Δ to vector index 'idx'
2:      for all col ∈ $[0, q\log(1/\delta))$ do
3:          col_hash ← hash(col, idx)
4:          row ← 0
5:          checksum ← $r[\text{col}]^{\text{idx}} \mod p$
6:          while row == 0  OR  col_hash[row-1] == 0 do
7:              col[row].a ← col[row].a + idx × Δ
8:              col[row].b ← col[row].b + Δ
9:              col[row].c ← col[row].c + Δ × checksum
10:             row ← row + 1
11: function QUERY_SKETCH( )                                        ▷ Get a non-zero vector index
12:     for all col ∈ $[0, q\log(1/\delta))$ do
13:         for all bucket ∈ col do
14:             value ← bucket.a/bucket.b
15:             if value is integer  AND  bucket.c == bucket.b × $r[\text{col}]^{\text{value}} \mod p$ then
16:                 return {value, bucket.b}                        ▷ Found a good bucket, done
17:     return sketch_failure                                       ▷ All buckets bad
```

Fig. 3. State—of—the—art ℓ_0-sampling algorithm.

Existing ℓ_0-samplers are slow to update for graph streaming workloads. Note in line 9 of Figure 3 that updating $c_{i,j}$ of bucket $\mathcal{B}_{i,j}$ requires modular exponentiation (computed on line 5), necessitating $O(\log(n))$ multiplication operations and $O(\log(n))$ modulo operations (where the modulus is a large prime). As a result, in the worst case, this algorithm performs $O(\log(n)\log(1/\delta))$ arithmetic operations per stream update. In the average case, the update modifies only $O(\log(1/\delta))$ buckets, however, the cost to generate checksums is still $O(\log(n)\log(1/\delta))$. Moreover, for sufficiently large vectors, this modular exponentiation must be done on integers larger than a 64-bit machine word, drastically increasing computation time in practice.

The "Standard ℓ_0" column of Figure 4 displays the single-threaded ingestion rate in updates per second of the state-of-the-art ℓ_0-sampling algorithm for vectors of various sizes. These results were obtained on a Dell Precision 7820 with 24-core 2-way hyperthreaded Intel(R) Xeon(R) Gold 5220R CPU @ 2.20 GHz and 64 GB 4x16 GB DDR4 2933 MHz RDIMM ECC Memory. Note how

Vector Length	Standard ℓ_0	CubeSketch	Speedup
1e3	2.2e+5	7.3e+6	33×
1e4	1.2e+5	5.1e+6	42×
1e5	5.4e+4	4.3e+6	80×
1e6	2.9e+4	3.7e+6	130×
1e7	2.0e+4	3.1e+6	150×
1e8	1.6e+4	2.8e+6	170×
1e9	1.3e+4	2.5e+6	200×
1e10	1.3e+3	2.2e+6	1700×
1e11	920	2.1e+6	2300×
1e12	830	1.9e+6	2300×

Fig. 4. CubeSketch is faster than standard ℓ_0 sketching. Ingestion rates (in updates/second) are listed for both ℓ_0 sketching methods.

Vector Length	Standard ℓ_0	CubeSketch	Size Reduction
1e+3	2.30KiB	1.21KiB	1.9×
1e+4	4.98KiB	2.34KiB	2.1×
1e+5	7.23KiB	3.43KiB	2.1×
1e+6	9.90KiB	4.73KiB	2.1×
1e+7	14.1KiB	6.79KiB	2.1×
1e+8	17.8KiB	8.58KiB	2.1×
1e+9	21.9KiB	10.6KiB	2.1×
1e+10	55.9KiB	13.6KiB	4.1×
1e+11	66.0KiB	16.1KiB	4.1×
1e+12	77.0KiB	18.8KiB	4.1×

Fig. 5. CubeSketch is significantly smaller than standard ℓ_0 sketching. Sizes are listed for both ℓ_0 sketching methods.

ingestion rate decreases as vector length increases, and in particular there is a catastrophic slowdown at vector length 10^{10}. This dramatic decrease in ingestion rate is due to the need to perform modular exponentiation on integers larger than 2^{64}, requiring the use of 128-bit integers thus slowing computation. When sketching characteristic vectors of length $O(V^2)$ for streaming connected components, 128-bit integers are required when $V \geq 10^5$.

When using ℓ_0-sampling for Boruvka emulation, each stream update $((u, v), \Delta)$ must be applied to the node sketches of u and v. For any node u, the node sketch of u is made up of $\log(V)$ ℓ_0-sketches of a_u. Each of these ℓ_0-sketches has a failure rate of $\delta = 1/100$ and, therefore, a width of $\log(1/\delta) = 7$. Processing a stream update requires $2 \cdot 7 \cdot O(\log^2(|a_u|) = 28 \cdot O(\log^2(V))$ multiplication and modulo operations. For a graph with a million nodes, StreamingCC must apply each update to 28 sketch vectors of length 10^{12}, so it can process roughly $800/28 = 29$ edge updates per second.

Existing ℓ_0-samplers are large for graph streaming workloads. Each node sketch consists of $\log(V)$ ℓ_0 sketches and each ℓ_0-sketch is a vector of $7c\log(V^2) = 14c\log(V)$ buckets. Each bucket is composed of three integers so a node sketch consists of $42c\log^2(V)$ integers. As noted above, 128-bit(16 B) integers are necessary when $V \geq 10^5$, so for $c = 2$ the size of a node sketch is $1344\log^2(V)$B. Since there is a node sketch for each node in the graph, the entire streaming data

	Standard ℓ_0		CubeSketch		
Vector Length	Mean	St.Dev.	Mean	St.Dev.	Speedup
2e+4	1.4e+6	1.7e+6	1.1e+7	6.8e+6	7.9×
1e+5	7.4e+5	2.1e+5	1.4e+7	5.9e+6	18×
1e+6	5.4e+5	1.2e+5	6.9e+6	2.4e+6	13×
1e+7	4.7e+5	1.0e+5	1.3e+7	5.2e+6	28×
1e+8	3.4e+5	1.1e+5	1.5e+7	4.9e+6	44×
1e+9	3.3e+5	8.1e+4	1.1e+7	5.3e+6	33×
1e+10	6.6e+4	2.1e+4	1.1e+7	5.7e+6	170×
1e+11	4.5e+4	1.1e+4	1.2e+7	5.5e+6	270×
1e+12	3.8e+4	1.2e+4	1.2e+7	6.1e+6	320×

Fig. 6. CubeSketch answers queries faster than standard ℓ_0 sketching. Query speeds (in queries/second) are listed for both ℓ_0 sketching methods.

structure has size $1344V\log^2(V)$B. When $V = 1$ million, this data structure is roughly 500 GiB in size.

Existing ℓ_0-samplers are slow to query for graph streaming workloads. Querying an ℓ_0 sketch requires performing a modular exponentiation, at a cost of $O(\log(n))$ multiplications, for each bucket. Figure 6 shows that, for a vector of length 10^{12} (corresponding to a graph with 1 million nodes), a ℓ_0 sketch query takes 1/38000 of a second or 26 microseconds. The first Boruvka round of a connectivity query for a graph on 1 million nodes requires querying 1 million ℓ_0 sketch sketches, which takes $1e+6/38000 > 26$ seconds, in addition to the cost of merging the sketches. A system operating under these parameters would be limited to at most 2 queries per minute. A user that wants to make frequent connectivity queries on a high-speed graph stream would find this lower bound on query latency prohibitively high.

Using existing ℓ_0-samplers offers no advantage on modern hardware. The goal of a streaming connected components algorithm is to use smaller space than would be required to store the entire graph explicitly. As we demonstrate empirically in Section 6, the most space-efficient dynamic graph processing system, Aspen, requires roughly 4 B of space for each edge in the graph. A straightforward back-of-the-envelope calculation reveals that even for dense graphs with average degree $V/2$, StreamingCC would use less space than Aspen only on very large inputs which require enormous RAM capacities and decades of processing time: $1344V\log^2(V)$B $\leq 4B \cdot V^2/4$ only when $V \geq 5 \cdot 10^5$. Processing half a million-node graph using StreamingCC would require 220 GB of RAM and, at an ingestion rate of less than 35 edges per second, would take more than 56 years to process the graph's roughly 64 billion edges. While StreamingCC's space complexity is much smaller than explicit graph representations like Aspen's asymptotically, in absolute terms it offers no advantage on modern hardware.

3.1 Improved ℓ_0-Sampler for Graph Connectivity

We present CubeSketch, an ℓ_0-sampling algorithm for vectors on the integers mod 2, which is smaller than the best existing general-purpose ℓ_0-sampling algorithm and is asymptotically faster to update. Since addition of characteristic vectors (Section 2.2) can be thought of as addition over vectors $\in \mathbb{Z}_2$, CubeSketch is sufficient for solving the connected components problem. Additionally, CubeSketch may be useful for other sketching algorithms for problems such as edge- or

```
function UPDATE_SKETCH(idx)                                ▷ Toggle vector index ‘idx’
    for all col ∈ $[0, q\log(1/\delta)$ do
        col_hash ← hash$_1$(col, idx)
        row ← 0
        checksum ← hash$_2$(col, idx)
        while row == 0   OR   col_hash[row-1] == 0 do
            col[row].$\alpha$ ← col[row].$\alpha$ ⊕ idx
            col[row].$\gamma$ ← col[row].$\gamma$ ⊕ checksum
            row ← row + 1
function QUERY_SKETCH( )                                   ▷ Get a non-zero vector index
    for all col ∈ $[0, q\log(1/\delta))$ do
        for all bkt ∈ col do
            if bkt.$\gamma$ == hash$_2$(col, bkt.$\alpha$) then
                return bkt.$\alpha$                         ▷ Found a good bucket, done
    return sketch_failure                                  ▷ All buckets bad
```

Fig. 7. Pseudocode for the CubeSketch algorithm.

vertex-connectivity, testing bipartiteness, and finding minimum spanning trees and densest subgraphs [2, 3, 30, 57].

Since CubeSketch's goal is to recover a nonzero entry from vectors of integers mod 2, it can use a much simpler bucket data structure than the general-purpose ℓ_0-sketch, improving space and update time costs. The CubeSketch algorithm is summarized in Figure 7. Each bucket $\mathcal{B}_{i,j}$ maintains two values: $\alpha_{i,j}$, which is used to recover the position of a single nonzero entry, and $\gamma_{i,j}$, which is used as a checksum. $\alpha_{i,j}$ and $\gamma_{i,j}$ are each $O(\log(n))$ bits, and, therefore, require $O(1)$ machine words. Since each vector value is either 0 or 1, $\Delta = 1$ for every stream update (e, Δ), and so for simplicity we refer to the update as (e).

Function UPDATE_SKETCH() in Figure 7 describes how CubeSketch processes a stream update. Given update (e), if $h_1(e) \equiv 0 \pmod{2^i}$ then e is in $\mathcal{B}_{i,j}$. For each such $\mathcal{B}_{i,j}$, $\alpha_{i,j} = \alpha_{i,j} \oplus bin(e)$ and $\gamma_{i,j} = \gamma_{i,j} \oplus h_2(bin(e))$ where $\oplus$ denotes bitwise XOR, $bin(e_w)$ denotes the binary representation of e_w, and h_1 and h_2 are hash functions drawn from a 2-wise independent family of hash functions. Note that the procedure for determining whether $e \in \mathcal{B}_{i,j}$ is identical to the algorithm in Figure 3, but the procedure for updating $\mathcal{B}_{i,j}$ is different. Importantly, CubeSketch never performs modular exponentiation, which as we will show makes it a $log(V)$ factor faster than the existing algorithm in the average case. As a result of UPDATE_SKETCH(), given a sequence of updates $(e_1), (e_2), \ldots, (e_k)$ to the data structure

$$\alpha_{i,j} = \bigoplus_{w \in [k]} bin(e_w), \tag{1}$$

$$\gamma_{i,j} = \bigoplus_{w \in [k]} h_2(bin(e_w)). \tag{2}$$

Function QUERY_SKETCH() describes how CubeSketch returns a nonzero entry of the input vector. For any bucket $\mathcal{B}_{i,j}$:

$$result = \begin{cases} e' & \text{if } \alpha_{i,j} = bin(e') \text{ and } \gamma_{i,j} = h_2(bin(e')) \\ \text{FAIL} & \text{if } \alpha_{i,j} = 0 \text{ and } \gamma_{i,j} = 0 \text{ OR} \\ & \text{if } \gamma_{i,j} \neq h_2(\alpha_{i,j}) \end{cases}$$

A nonzero entry is recovered from CubeSketch by attempting to recover a nonzero entry from each $\mathcal{B}_{i,j}$ until one returns a value other than FAIL. If no such bucket exists, the algorithm returns NULL.

Theorem 1. *CubeSketch is an ℓ_0 sampler that, for input vector $x \in \mathbb{Z}_2^n$, has space complexity $O(\log^2(n)\log(1/\delta))$, worst-case update complexity $O(\log(n)\log(1/\delta))$, average-case update complexity $O(\log(1/\delta))$, and failure probability at most δ.*

Proof. The space and update time results follow by construction: each bucket $\mathcal{B}_{i,j}$ requires a constant number of machine words, and $i \in [O(\log(n)])$ and $j \in [O(\log(1/\delta)]$. Applying an update to any bucket $\mathcal{B}_{i,j}$ requires constant time, and in the worst case, an update will be applied to each of the $O(\log(n)\log(1/\delta))$ buckets. In the average case, an update is applied to $O(\log(1/\delta))$ buckets.

Lemma 2. *CubeSketch's selection process succeeds with probability at least $1-\delta$. Equivalently, CubeSketch contains a bucket $\mathcal{B}_{i,j}$ with a single nonzero entry, that is,* $\Pr\left[\exists i, j \text{ s.t. } supp(\mathcal{B}_{i,j}) = 1\right] \geq 1-\delta$.

Proof. Adapted from [21]. Choose $i \in [\log(n)]$ such that $2^{i-2} \leq \|x\|_0 < 2^{i-1}$ where $\|x\|_0$ denotes the ℓ_0 norm of x, i.e., the number of nonzero entries of x. Let A_x be the set of positions of nonzero entries in x. Then, since h_1 is drawn from a 2-universal family of hash functions, $\forall j \in [6\log(1/\delta)]$,

$$\begin{aligned}\Pr\left[supp(\mathcal{B}_{i,j} = 1)\right] &= \sum_{k \in A_x} \frac{1}{2^i}\left(1 - \frac{1}{2^i}\right)^{\|x\|_0 - 1} \\ &> \frac{\|x\|_0}{2^i}\left(1 - \frac{\|x\|_0}{2^i}\right) > 1/8.\end{aligned}$$

Then $\Pr\left[supp(\mathcal{B}_{i,j} \neq 1) \forall j \in [6\log(1/\delta)]\right] < (1-1/8)^{6\log(1/\delta)} = (7/8)^{6\log_{7/8}(1/\delta)/\log_{7/8}(2)} = \delta^{-6/\log_{7/8}(2)} < \delta$. □

Lemma 3. *CubeSketch's checksum succeeds with high probability. That is, $\forall w, y$, if $supp(\mathcal{B}_{w,y}) = 1$ then $\gamma_{w,y} = h_2(\alpha_{w,y})$ and if $supp(\mathcal{B}_{w,y}) > 1$ then $\Pr\left[\gamma_{w,y} \neq h_2(\alpha_{w,y})\right] \geq 1 - 1/n^c$ for some constant c.*

Proof. When $\mathcal{B}_{i,j}$ has a single nonzero entry, it always passes the error check. That is, if $supp(\mathcal{B}_{i,j}) = 1$, $\alpha_{w,y} = bin(e_i)$ where e_i is the single nonzero element of $\mathcal{B}_{i,j}$, and $\gamma_{w,y} = h_2(bin(e_i))$.

When $\mathcal{B}_{i,j}$ has more than one nonzero entry, then it passes the error check only in the rare event of a hash collision: If $supp(\mathcal{B}_{i,j}) > 1$, fix $e_i \in \mathcal{B}_{i,j}$. By Equations (1) and (2), $\gamma_{w,y} = h_2(\alpha_{w,y})$ iff $\bigoplus_{j \in \mathcal{B}_{i,j} \setminus e_i} h_2(bin(j)) \oplus h_2(bin(e_i)) = h_2(\alpha_{w,y})$. Since h_2 is a 2-wise independent hash function, assuming that $\gamma_{i,j}$ is $c\log(n)$ bits:

$$\Pr\left[h_2(bin(e_i)) = \left(\bigoplus_{j \in \mathcal{B}_{i,j} \setminus e_i} h_2(bin(j))\right) \oplus h_2(\alpha_{w,y})\right] = \frac{1}{2^{c\log(n)}} = \frac{1}{n^c}.$$

□

Lemmas 2 and 3 imply that CubeSketch is sampleable with probability $1-\delta$ (see Definition 1). CubeSketch may be added via elementwise $\bigoplus$ (exclusive or). Linearity of CubeSketch follows from the observation that exclusive or is a linear operation. □

Figure 4 illustrates that CubeSketch is far faster than the standard ℓ_0-sampling algorithm. In fact, when sketching characteristic vectors of graphs with at least 10^5 nodes, it is more than 3 orders of magnitude faster. This dramatic speedup is a result both of CubeSketch's asymptotically lower update time complexity, and the fact that its update cost is dominated by bitwise exclusive OR operations, which are in practice much faster than the division operations standard ℓ_0-sampling performs. Similarly, CubeSketch's query operations require computing one hash per bucket, which is fast in practice. Finally, standard ℓ_0 sampling is slowed significantly by the need to perform $O(\log(V)\log(1/\delta)$ modular exponentiation operations on 128-bit integers for each update when $V \geq 10^5$. CubeSketch does not require 128-bit operations until processing graphs with tens of billions of nodes.

Figure 5 shows that, for the same input vector length and failure probability, CubeSketch is twice as small as standard ℓ_0 sampling for smaller vectors and four times smaller for larger vectors. This is a result of the fact that CubeSketch's bucket data structures use half the machine words of standard ℓ_0 sampling and the fact that CubeSketch does not need to use 128-bit integers for longer vectors.

Querying CubeSketch is up to 320 times faster than standard ℓ_0 sketching due in part to eliminating modular exponentiation and requiring only 64 bit machine words (as for ingestion above). Per Figure 6, on a million-node graph CubeSketch performs 12 million queries per second or only 83 nanoseconds per query. When performing the first round of a connectivity query on a graph with 1 million nodes, we query all the sketches which takes only ≈ 83 milliseconds. This is much faster than the 26 seconds required for the standard ℓ_0 sketch as described above. CubeSketch's low query latency allows it to quickly answer queries, even on large datasets. In Section 6.6 we note that GraphZeppelin's connectivity query performance depends on both the underlying sketches' ingestion *and* query performance.

4 BUFFERING FOR I/O EFFICIENCY AND IMPROVED PARALLELISM

In the streaming connectivity problem, stream updates are *fine-grained*: each update represents the insertion or deletion of a single edge. Since streams are ordered arbitrarily, even a short sequence of stream updates can be highly non-local, inducing changes throughout the graph. As a result, StreamingCC and similar graph streaming algorithms do not have good data locality in the worst case. This lack of locality can cause many CPU cache misses and therefore reduce the ingestion rate, even when sketches are stored in RAM. The cache-miss cost can be high since ingesting each stream update (u, v, Δ) requires modifying a logarithmic number of sketches, and can thus induce a poly-logarithmic number of cache misses. The consequences are even worse if sketches are stored on disk since each edge update requires loading a logarithmic number of sketches from disk, leading to the following observation.

Observation 1. *In the hybrid semi-streaming model with $M = o(V\log^3(V))$ RAM and $D = \Omega(V\log^3(V))$ disk, StreamingCC uses $\Omega(1)$ I/Os per update and processing the entire stream of length N uses $\Omega(N) = \Omega(E)$ I/Os.*

Any sketching algorithm that scales out of core suffers severe performance degradation unless it amortizes the per-update overhead of accessing disk. Such an amortization is not straightforward, since sketching inherently makes use of hashing and as a result induces many random accesses, which are slow on persistent storage. We now introduce a sketching algorithm for the streaming connected components problem that amortizes disk access costs, even on adversarial graph streams, and as a result is simultaneously a space-efficient graph semi-streaming algorithm and an I/O-efficient external-memory algorithm. We also note that the design facilitates parallelism, which we experimentally verify in Section 6.

4.1 I/O-Efficient Stream Ingestion

We describe GraphZeppelin's I/O efficient stream ingestion procedure in the hybrid streaming model (see Section 2.1).

Arbitrarily partition the nodes of the graph into ***node groups*** of cardinality $\max\{1, B/\log^3(V)\}$. Let $\mathcal{U} \subset \mathcal{V}$ denote a node group, and let $\mathcal{S}(\mathcal{U})$ denote the node sketches associated with the nodes in $\mathcal{U}$. Store $\mathcal{S}(\mathcal{U})$ contiguously on disk. This allows $\mathcal{S}(\mathcal{U})$ to be read into memory I/O efficiently: if node groups are of cardinality 1, then B is smaller than the size of a node sketch, and if each node group has cardinality $B/\log^3(V) > 1$, then the sketches for the group have total size $O(B)$.

Applying stream update $((u, v), \Delta)$ to node sketches of u and v immediately upon arrival takes $\Omega(1)$ I/Os since the corresponding sketches must be read from disk. To amortize the cost of fetching sketches, GraphZeppelin only fetches $\mathcal{S}(\mathcal{U}_i)$ when it has collected $\max\{B, \log^3(V)\}$ updates for $\mathcal{U}_i$. Since there may be $O(V)$ node groups, collecting these updates for each node group cannot be done in RAM. Instead, we collect these updates I/O efficiently on disk using a ***gutter tree***, a simplified version of a buffer tree [9] which uses $O(V(\log^3(V))$ space.

Like a buffer tree, a gutter tree consists of a tree whose vertices each have buffers of size $O(M)$. Each non-leaf vertex has $O(M/B)$ children. We refer to a leaf vertex of the gutter tree as a ***gutter***, because it fills with stream data but is periodically emptied by applying the contained stream data to sketches. Each leaf vertex in the gutter tree is associated with a node group $\mathcal{U}$ and has size $\max\{B, \log^3(V)\}$, the same size as $\mathcal{S}(\mathcal{U})$. When a gutter for node group $\mathcal{U}$ fills, GraphZeppelin reads $\mathcal{S}(\mathcal{U})$ and the updates stored in the gutter into memory, applies the updates to $\mathcal{S}(\mathcal{U})$, and writes $\mathcal{S}(\mathcal{U})$ back to disk. Since data does not persist in leaf vertices, no rebalancing is necessary.

Lemma 4. *GraphZeppelin's stream ingestion uses $O(V \log^3(V))$ space and $sort(N) = O(N/B(\log_{M/B}(V/B)))$ I/Os in the hybrid streaming setting.*

Proof. GraphZeppelin's sketch data structures use $O(V \log^3(V))$ space.

Each leaf in the gutter tree has a gutter of size $\max\{B, \log^3(V)\}$. This is one gutter for each node group and there are $V/(\max\{1, B/\log^3(V)\})$ node groups so the total space for the leaves of the gutter tree is $O(V \log^3(V))$.

In the level above the leaves, there are $V \log^3(V)/B \cdot B/M$ vertices each with size M, so the total space used at this level is $O(V \log^3(V))$. Each subsequent higher level of the tree uses $O(M/B)$ space less than the level below it, so the total space used for the entire gutter tree is $O(V \log^3(V))$.

The I/O complexity of the gutter tree is equivalent to that of the buffer tree, except that leaf gutters are flushed by reading in the appropriate sketches from disk and applying the updates in the gutter to these sketches. Asymptotically this incurs no additional cost so the total I/O complexity for ingestion is $sort(N)$. ◻

4.2 I/O-Efficient Connectivity Computation

Lemma 5. *Once all stream updates have been processed, GraphZeppelin computes connected components using $O((V \log^3(V)/B) + (V \log^*(V))$ I/Os in the hybrid streaming model.*

Proof. Each round of Boruvka's algorithm has three phases. In the first, an edge is recovered from the sketch of each current connected component. In the second, for each edge its endpoints are merged in a disjoint set union data structure which keeps track of the current connected components. In the third phase, for each pair of connected components merged in phase 2, the corresponding sketches are summed together. We analyze the I/O cost of each phase of a round separately.

In the first round, to query the sketches in the first phase, all of the sketches must be read into RAM which can be done with a single scan. This uses $O(V \log^3(V)/B)$ I/Os.

The disjoint set union data structure has size $O(V)$ and must be stored on disk. In the second phase, the cost of each DSU merge is $\log^*(V)$ I/Os, because a merge requires a leaf-to-root traversal of the union find data structure and this leaf-to-root path has length at most $\log^*(V)$. In the worst case, each parent resides in a different block from its child so each step of the path requires an I/O. Since there are at most V merges, the total I/O cost is $V \log^*(V)$.

In the third phase, summing the sketches of the merged components together is I/O efficient if $B = O(\log^3(V))$, since the disk reads and writes necessary for summing sketches are the size of a block or larger. The cost for the third phase is $O(V \log^3(V)/B)$.

If $B = \omega(\log^3(V))$, sketches are much smaller than the block size. Since the merges performed in each round of Boruvka are a function both of the input stream and of the randomness of the sketches, these merges induce random accesses to the sketches on disk and so summing the sketches for each merge takes $O(1)$ I/Os. In total, the third phase takes $O(V)$ I/Os in this case.

Since the number of connected components decreases by at least half in each round, the I/O cost of each round at at most half the cost of the previous round. Therefore, the asymptotic cost of the entire Boruvka algorithm is the cost of the first round, that is, $O((V \log^3(V)/B) + (V \log^*(V))$ I/Os. □

Corollary 1. *When $E = \Omega(V \log^3(V))$ and $B = o(\log^3(V))$ or $M = O(V)$, GraphZeppelin is I/O optimal for the connected components problem; i.e., it uses $sort(E) = O(E/B(\log_{M/B}(V/B)))$ I/Os.*

Note that for optimality the graph cannot be too sparse. In practice, for some graph streams $M = O(VB)$ and $D = O(V \log^3(V))$. In this case, we can omit the upper levels of the gutter tree and write I/O efficiently to the leaf gutters stored on disk. In Section 5, we describe how GraphZeppelin can perform stream ingestion using either a full gutter tree or just the leaf gutters, and evaluate the performance of both approaches in Section 6.

5 SYSTEM DESCRIPTION

The GraphZeppelin algorithm is split into two components: ***stream ingestion***, in which edge updates are processed and stored using CubeSketch, and ***query-processing***, in which a spanning forest for the graph is recovered from these sketches. These components use SSD when the sketches are so large that they do not fit in RAM. Their implementations are parallel for better performance on multi-core systems.

GraphZeppelin's user-facing API consists of edge_update() for processing stream updates, and list_spanning_forest() to compute and return the connected components. On initialization, GraphZeppelin allocates $\log(V)$ CubeSketch data structures for each node in the graph, for a total sketch size of approximately $280V \cdot \log^2(V)$ bytes. It also initializes its buffering data structure.

5.1 Stream Ingestion

Each update in the input stream is immediately placed into a buffering system. Periodically, the buffering system produces a batch of updates bound for the same graph node u. This batch is inserted into a work queue, which then hands the batch off to a ***Graph Worker***, i.e., a thread for carrying out batched sketch updating. Because each batch is only applied to a single node sketch, and because each of the $\log(V)$ CubeSketches in a node sketch can be updated in parallel, many Graph Workers can operate in parallel without contention (see Section 5.1). A high-level illustration and pseudo code of GraphZeppelin stream ingestion are shown in Figures 8 and 9, respectively.

Buffering. GraphZeppelin's buffering system ingests updates from the stream and periodically outputs a batch of updates for a single node in the graph. GraphZeppelin implements two buffering data structures: a gutter tree, described in Section 4, and a simplified version of the gutter tree, which only includes the leaves. Depending upon available memory, GraphZeppelin uses only one of these two buffering structures at any time. The leaf-only version is fundamentally a special case gutter tree used when sufficient memory is available ($M > V \cdot B$) and is optimized for this case.

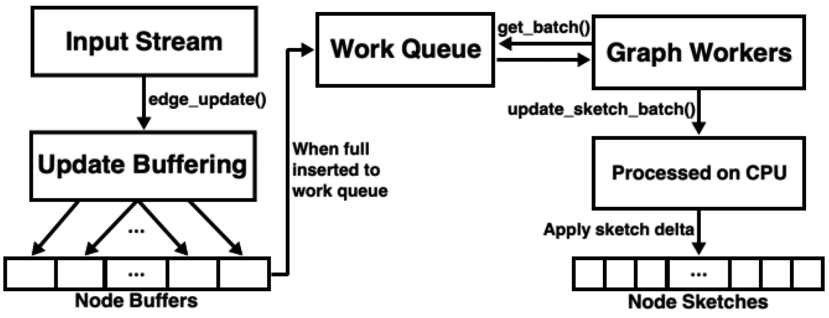


Fig. 8. GraphZeppelin stream ingestion data flow.

```
 1: function EDGE_UPDATE(edge ← {u, v})                ▷ Write edge update to buffers
 2:     BUFFER_INSERT({u, v})
 3:     BUFFER_INSERT({v, u})
 4: function DO_BATCH_UPDATE( )                        ▷ Apply batched updates to supernode
 5:     {batch, node} ← GET_BATCH( )
 6:     for all sketch ∈ supernodes[node] do
 7:         UPDATE_SKETCH_BATCH(sketch, batch)
 8: function UPDATE_SKETCH_BATCH(sketch, batch)
 9:     for all update ∈ batch do
10:         sketch.UPDATE_SKETCH(update)
```

Fig. 9. Pseudocode for GraphZeppelin's core stream ingestion routines. EDGE_UPDATE() is part of the user API, while DO_BATCH_UPDATE(), and UPDATE_SKETCH_BATCH() are internal functions.

These buffering techniques confer several benefits. First, when GraphZeppelin's sketches are so large that they do not fit in RAM and are stored on SSD, applying updates to a single node sketch in large batches amortizes the I/O cost of reading the node sketch into memory. Without buffering, each stream update would incur $\Omega(1)$ I/Os in the worst case. We demonstrate in Section 6.4 that buffering facilitates I/O efficiency and parallelism.

Gutter tree. GraphZeppelin allocates 8MB for each non-leaf buffer in the gutter tree. The gutter tree writes updates to the disk in blocks of 16KB, and has a fan-out of $\frac{8\text{MB}}{16\text{KB}} = 512$. A write block of 16KB is an efficient I/O granularity for SSDs and a buffer size of 8MB balances buffering performance with the latency of flushing updates through the gutter tree. When $V > 5 \cdot 10^4$, the size of a sketch is greater than 100KB, much larger than the 16KB block. Therefore, the leaf nodes of the gutter tree accumulate updates for a single graph node. GraphZeppelin allocates space for each leaf gutter equal to twice the size of a node sketch.

When we initialize GraphZeppelin, we leverage the static structure of the gutter tree to pre-allocate its disk space. A call to BUFFER_INSERT($\{u, v\}$) inserts $\{u, v\}$ to the root buffer of the gutter tree. Another thread asynchronously flushes the contents of full buffers to the appropriate child using the *pwrite* system call. When a flush causes the buffer of a child node to fill, that child node is recursively flushed before the flush of the parent continues. When a leaf gutter is full this thread moves the batch of updates into the work queue for processing by Graph Workers in DO_BATCH_UPDATE().

Leaf-only gutter tree. For each graph node u we maintain a gutter that accumulates updates for u. When the system is initialized, we allocate the memory for each of these gutters. By default, each leaf gutter is 1/2 the size of a node sketch. This choice balances RAM usage with I/O efficiency as shown in Section 6.4. BUFFER_INSERT((u, v)) inserts edge $e = (u, v)$ directly into the gutter for node

u. As before, when the gutter becomes full, it is flushed and the batch is inserted into the work queue. Note that the leaf-only gutter data structure need not fit entirely in RAM, so long as at least a page of memory is available per buffer the rest can be efficiently swapped to SSD; see Section 6.

Work queue. The work queue functions as a simple solution for the producer-consumer problem, in which the thread filling buffers produces work and the Graph Workers consume it. Once a buffer is filled the BUFFER_INSERT() function inserts the batch of updates into the work queue. Later a Graph Worker removes the batch from the front of the queue in DO_BATCH_UPDATE().

Insertions to the queue are blocked while the queue is full, and Graph Workers in need of work are blocked while the queue is empty. The work queue can hold up to $8g$ batches, where g is the number of Graph Workers. A moderate work queue capacity of $8g$ limits the time either the buffering system or graph workers spend waiting on the queue, even when batch creation is volatile, while keeping the memory usage of the work queue low.

Sketch updates. In each call to DO_BATCH_UPDATE(), Graph Workers call GET_BATCH() to receive a batch of updates bound for a particular node u from the work queue. The Graph Worker then uses UPDATE_SKETCH_BATCH(sketch$_u$, batch) to update each of the $O(\log(V))$ CUBESKETCHES in the node sketch of u.

As described in Section 3.1, a CUBESKETCH is a vector of buckets, each of which consists of a 64 bit α value and a 32 bit γ value. Each CUBESKETCH stores a two dimensional array A of buckets $\mathcal{B}_{i,j}$, with dimensions $\log(V^2) \times (\log(1/\delta) = 7)$. To apply an update $(e = \{u, v\})$ to a CUBESKETCH, the Graph Worker determines which buckets $\mathcal{B}_{i,j}$ contain e, and sets $\alpha_{i,j} := \alpha_{i,j} \oplus e$ and $\gamma_{i,j} := \gamma_{i,j} \oplus h_y(e)$. The hash values are calculated using xxHash [20].

Each CUBESKETCH data structure uses $7\log(V^2) = 14\log(V)$ 12 B buckets. In total, this is $168\log(V)$ bytes per CUBESKETCH, and $168\log(V)\log_{3/2}(V)$ bytes per node sketch.

Multithreading sketch updates. Applying a batch to a node sketch in DO_BATCH_UPDATE() is handled asynchronously by a Graph Worker, allowing what we call ***batch-level parallelism***. We implement these workers using C++ STL threads.

We use OpenMP [65] to dispatch a group of threads to process each CUBESKETCH update in UPDATE_SKETCH_BATCH(). We refer to this as ***sketch-level parallelism***. OpenMP allows us to specify the number of threads to allocate to a task and handles work allocation transparently. When updating a node sketch, applying a batch to each CUBESKETCH is treated as one work unit and OpenMP allocates the $\log(V)$ units between the apportioned threads.

Implementing both batch- and sketch-level parallelism gives us a natural way to tune GRAPHZEPPELIN's performance. For instance, we can decide to configure more Graph Workers with fewer threads per group, or fewer Graph Workers with more threads per group. We experimentally determine a good configuration for our hardware and datasets (see Section 6.4).

A single work unit is never shared between threads in the same group. As a result, a CUBESKETCH is only modified by one thread in a group, so no locking is necessary at the sketch level. However, locking is necessary at the batch level because consecutive batch updates may be requested to the same node sketch, and thus multiple graph workers may seek to dispatch thread groups to the same sub-sketches. We minimize the size of this critical section by exploiting linearity of ℓ_0-samplers. Rather than locking a node sketch $S(x)$ for the entire batch operation, we apply the updates to an empty sketch $S(x_0)$ and lock only to add $S(x) = S(x) + S(x_0)$.

5.2 Query Processing

When a connectivity query is issued, GRAPHZEPPELIN calls LIST_SPANNING_FOREST() which returns a spanning forest of the graph. The first step of post-processing is to flush the buffering data

```
1: function EDGE_UPDATE(edge ⟵ {u, v})                ▹ Write edge update to buffers
2:     BUFFER_INSERT({u, v})
3:     BUFFER_INSERT({v, u})
4: function DO_BATCH_UPDATE( )                         ▹ Apply batched updates to supernode
5:     {batch, node} ⟵ GET_BATCH( )
6:     for all sketch ∈ supernodes[node] do
7:         UPDATE_SKETCH_BATCH(sketch, batch)
8: function UPDATE_SKETCH_BATCH(sketch, batch)
9:     for all update ∈ batch do
10:        sketch.UPDATE_SKETCH(update)
```

Fig. 10. Pseudocode for GraphZeppelin's core post-processing routines. list_spanning_forest() is part of the user API, while cleanup() is an internal function.

structure of any remaining updates, moving the batches to the work queue in cleanup(). We then wait for the Graph Workers to finish processing these batches. Finally, GraphZeppelin runs Boruvka's algorithm to generate a spanning forest of the input graph. This algorithm is summarized in Figure 10.

6 EVALUATION

Experimental setup. We implemented GraphZeppelin as a C++14 executable compiled with g++ version 9.3 for Ubuntu. All experiments were run on a Dell Precision 7820 with 24-core 2-way hyperthreaded Intel(R) Xeon(R) Gold 5220R CPU @ 2.20 GHz, 64 GB 4x16 GB DDR4 2933 MHz RDIMM ECC Memory and two 1 TB Samsung 870 EVO SSDs. In some of our experiments, we artificially limited RAM to force systems to page to disk using Linux Control Groups. We put a swap partition and the gutter tree data on one of the two SSDs, and the other SSD held the datasets.

6.1 Datasets

We used two types of datasets in this article. First, we generated large, dense graphs using a Graph500 specification, and converted these to streams for our evaluation. We also evaluated correctness on graphs from the SNAP graph repository [48] and the Network Repository [70]. All datasets used are described in Figure 11.

Synthesizing Dense Graphs and Streams. We created undirected graphs using the Graph500 Kronecker generator. We produced five simple, undirected graphs. These graphs are dense: each has roughly one-half of all possible edges. The Graph500 generator does not output simple graphs by default, so to produce our five simple graphs we pruned duplicate edges and self-loops [8].

We then transformed each of the five graphs into a random stream of edge insertions and deletions with the following guarantees: (i) an insertion of edge e always occurs before a deletion of e, (ii) an edge never receives two consecutive updates of the same type, (iii) we disconnect a small (fewer than 150) set of nodes from the rest of the graph, and (iv) by the end of the stream, exactly the input graph (with the exception of the edges removed to disconnect the vertices in (iii)) remains. Note that this mechanism deliberately adds edges not in the original graph, but they are always subsequently deleted. We implemented (iii) to guarantee some non-trivial connected components in each stream's final graph.

Publicly Available Datasets. We also used the following real-world datasets. **p2p-gnutella** is a graph representing the Gnutella peer-to-peer network [69]. **rec-amazon** is a co-purchase recommendation graph for products listed on Amazon [47], where each node represents a product and

Name	# of Nodes	# of Edges	# Stream Updates	# Connected Components
kron13	2^{13}	1.7e+7	1.8e+7	26
kron15	2^{15}	2.7e+8	2.8e+8	51
kron16	2^{16}	1.1e+9	1.2e+9	76
kron17	2^{17}	4.3e+9	4.5e+9	101
kron18	2^{18}	1.7e+10	1.8e+10	126
p2p-gnutella	6.3e+4	1.5e+5	2.9e+5	12
rec-amazon	9.2e+4	1.3e+5	2.5e+5	2
google-plus	1.1e+5	1.4e+7	2.7e+7	4
web-uk	1.3e+5	1.2e+7	2.3e+7	2

Fig. 11. Dimensions of datasets used in this evaluation.

there is an edge between two nodes if their corresponding products are frequently purchased together. **google-plus** is a graph among users of the Google Plus social network [56] where edges represent follower relations. **web-uk** is a web graph, where edges represent links between pages [70]. Each of these real-world graphs was converted to a stream using the process described above.

6.2 GraphZeppelin is Fast and Compact

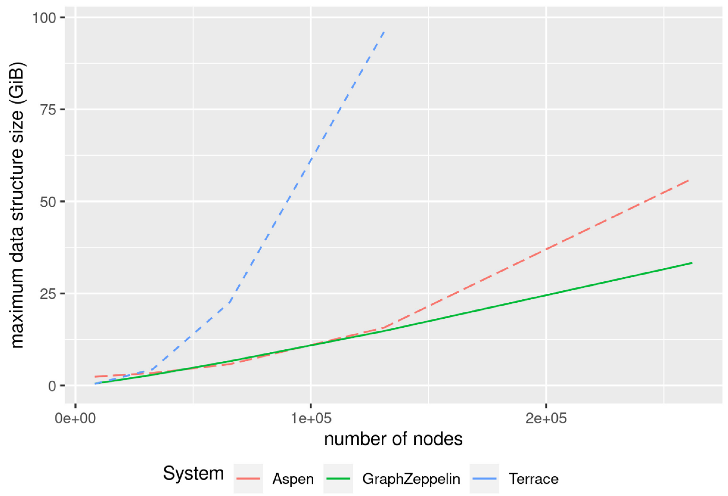
We now demonstrate that, given the same memory resources, GraphZeppelin can handle larger inputs than Aspen and Terrace on sufficiently large and dense graph streams. We also show that unlike these systems, GraphZeppelin maintains good performance when its data structures are stored on SSD.

Both Aspen and Terrace are optimized for the ***batch-parallel*** model of dynamic graph processing. In this model, updates are applied to a non-empty graph in batches containing exclusively insertions or exclusively deletions. This contrasts with our streaming model, an initially empty graph is defined entirely from a stream of interspersed inserts and deletes. To avoid unfairly penalizing Aspen and Terrace, we group the input stream into batches insertions and deletions to these systems (ignoring any query correctness issues this may introduce) and present these batches as the input stream. Whenever one of these arrays fills, we feed it into the appropriate batch update function provided by Aspen or Terrace.

We ran GraphZeppelin, Aspen, and Terrace on each Kronecker stream. We used a batch size of 10^6 for Aspen and Terrace because we found this to produce the highest ingestion rates for both systems. To record memory usage we logged the output of the Linux top command tracking each system every five seconds. All experiments were run for a maximum of 24 hours.

Memory Profiling. GraphZeppelin's space-efficient CubeSketches make it a $O(V/\log^3(V))$-factor smaller than Aspen or Terrace asymptotically. Given the polylogarithmic factors and constants, this experiment determines the actual crossover point where GraphZeppelin is more compact than Aspen and Terrace. As shown in Figure 12, GraphZeppelin is smaller than Terrace even on kron15, and uses roughly equivalent memory to Aspen on kron13–kron17. For kron18 we observe the Aspen uses roughly double the memory as GraphZeppelin. For larger dense graphs, we this difference will continue to grow because of the asymptotic difference in space usage.

Note that Terrace does not currently support batch deletions, so we rely on its individual edge deletion functionality instead and do not maintain a deletions array.



Dataset	Aspen	Terrace	GraphZeppelin
kron13	2.4	0.44	0.52
kron15	3.4	4.4	2.9
kron16	5.8	23	6.6
kron17	16	> 96	15
kron18	56	N/A	33
p2p-gnutella	2.2	0.009	4.2
rec-amazon	1.7	0.009	6.6
google-plus	2.5	0.3	8.3
web-uk	2.4	0.36	10

Fig. 12. GraphZeppelin uses less space than Aspen or Terrace to process large, dense graph streams. Space usage for each system is given in Gibibytes. Terrace timed out on kron17 in this experiment. For the sake of completeness, the table includes space utilization for all datasets. The chart includes only the dense Kronecker graphs.

I/O Performance and Ingestion Rate. Unlike Aspen and Terrace, GraphZeppelin maintains consistently high ingestion rates when its data structures are stored on SSD. In Figure 13 we summarize the results of running Aspen, Terrace, and GraphZeppelin with only 16GB of RAM. The ingestion rates of both Aspen and Terrace plummet once their data structures exceed 16GB in size and they are forced to store excess data on SSD. Neither Aspen nor Terrace were able to finish their largest evaluated stream within 24 hours (2^{17} for Terrace and 2^{18} for Aspen). In comparison, GraphZeppelin's ingestion rate remains high when its memory consumption extends into secondary storage. GraphZeppelin's gutter tree finished the kron18 stream with an average ingestion rate of 2.50 million updates per second, a 29% reduction to its performance compared to when its sketches are stored entirely in RAM.

In RAM, GraphZeppelin's ingestion rate is higher than Aspen's and Terrace's on all Kronecker streams. We summarize these results in Figure 14. Notably, on kron18 GraphZeppelin ingests 4.25 million updates per second, over three times faster than Aspen. GraphZeppelin ingests more than an order of magnitude faster than Terrace on these streams, so we omit it from the figure.

Figure 15 displays the ingestion rates of Aspen, Terrace, and GraphZeppelin on kron17 as a function of graph density. As with all of the Kronecker graph streams used in this work, the graph

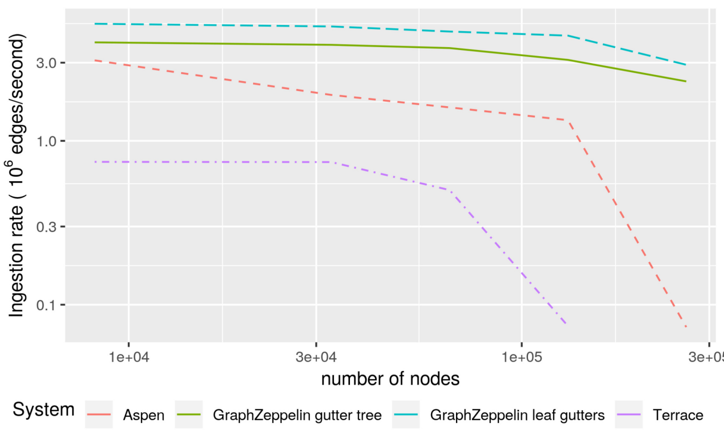


Fig. 13. GraphZeppelin remains fast even when its data structures are stored on disk, unlike Aspen and Terrace.

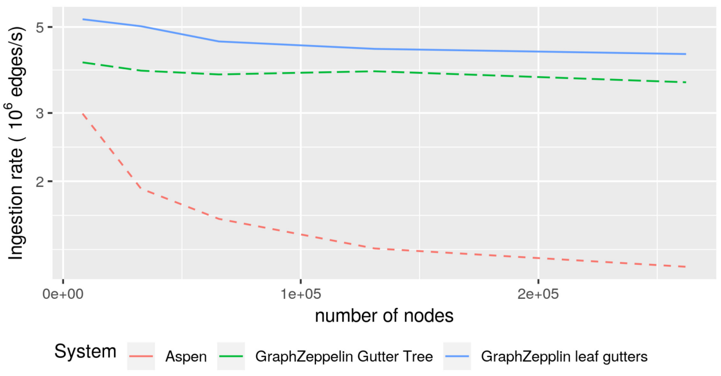


Fig. 14. GraphZeppelin is faster than Aspen and Terrace even when all data structures fit in RAM.

starts out empty at the beginning of the stream and gradually grows denser as the stream progresses and more edges are inserted. GraphZeppelin's 5 million updates/sec performance does not depend on graph density. In contrast, Aspen ingests quickly (almost 3 million u/s) at low density but quickly falls off to slightly more than 1 million u/s as density increases. Terrace ingests more slowly even when the graph is sparse and exhibits a slight fall-off as density increases. The drastic decrease in Terrace's ingestion rate at around 42.5% density is due to its data structures overflowing the available 64 GB of RAM and paging to disk (see Figure 13). The brief period of low (1.5 million u/s) ingestion rate for GraphZeppelin at very low density corresponds to the beginning of the stream, when GraphZeppelin is filling its buffers and not yet processing updates.

6.3 GraphZeppelin is Reliable

GraphZeppelin's sketching algorithm is not deterministically correct: it has a nonzero failure probability, which is guaranteed to be at most $1/V^c$ for some constant c. To establish that failures do not occur in practice, we compared GraphZeppelin with an in-memory adjacency matrix stored as a bit vector. Specifically, we applied stream updates to GraphZeppelin and the adjacency matrix and periodically queried GraphZeppelin and compared its results with the output of running Kruskal's algorithm on the adajacency matrix. We performed 1,000 such correctness checks

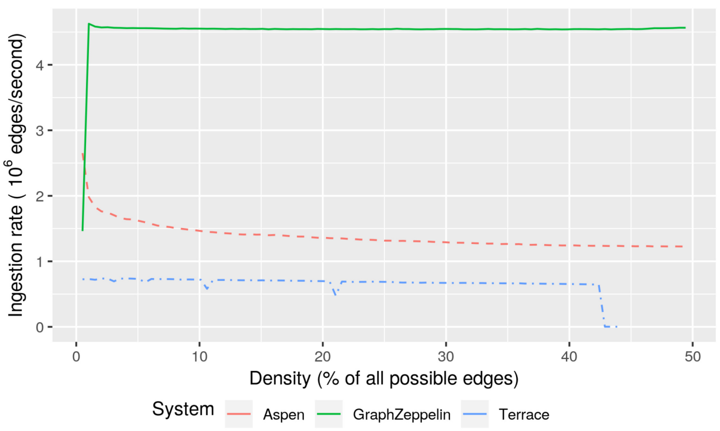


Fig. 15. GraphZeppelin's 5 million updates/sec performance does not depend on graph density. In contrast, Aspen ingests quickly (almost 3 million u/s) at low density but quickly falls off to slightly more than 1 million u/s as density increases. Terrace ingests more slowly even when the graph is sparse (likely due to unoptimized implementation of edge deletions) and exhibits a slight fall-off as density increases.

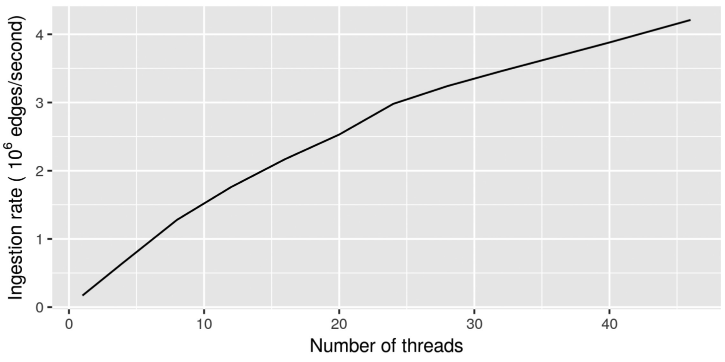


Fig. 16. GraphZeppelin updates sketches in parallel, increasing ingestion rate by 26× when using 46 threads.

each on the kron17, p2p-gnutella, rec-amazon, google-plus, and web-uk streams. No failures were ever observed. While our algorithm's performance is optimized for dense graphs, this experiment demonstrates that it succeeds with high probability for both dense and sparse graphs.

6.4 GraphZeppelin is Highly Parallel

Due to the atomized nature of sketch updates, we expect stream ingestion to scale well on multi-core systems. We experimentally demonstrate this claim by varying the number of threads used for processing updates and observe a significant speed-up.

Figure 16 shows the ingestion rate of GraphZeppelin as the number of threads processing the kron17 graph stream increases. The threads are given a pool of 64GB RAM so that the parallel performance can be measured without memory contention. To avoid external memory accesses, we use leaf-only gutters for buffering. The per-thread increase in ingestion rate is significant; the ingestion rate for 46 threads is approximately 26 times higher than that of a single thread. Additionally, at 46 threads the marginal ingestion rate is still positive, suggesting that adding more

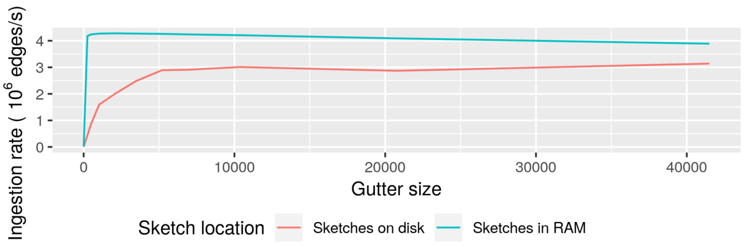


Fig. 17. GraphZeppelin gutter size vs. ingestion speed.

threads would further increase performance. We also experimentally determined that a group size of one gives the best performance with our combination of machines and inputs.

6.5 GraphZeppelin Buffering Facilitates Parallelism and I/O Efficiency

Applying sketch updates is highly scaleable, but only if updates are buffered and applied in batches. When sketches are stored on disk, processing each update individually requires $\Omega(1)$ IOs. Additionally, cache contention and thread synchronization bottleneck the ingestion rate even when sketches are in RAM. For these reasons we retain buffers of a constant factor f of the node-sketch size.

Figure 17 summarizes the ingestion rate of GraphZeppelin on the kron17 stream for different values of f when the sketches are stored in RAM and when they are stored on disk. GraphZeppelin is given 46 Graph Workers and a group size of 1. With buffers of size 1 (no buffering), GraphZeppelin ingests 130,000 updates per second in RAM, 33 times slower than when $f = .10$. On SSD, the ingestion rate is only 2,000 insertions per second, 3 orders of magnitude slower than peak on-disk performance.

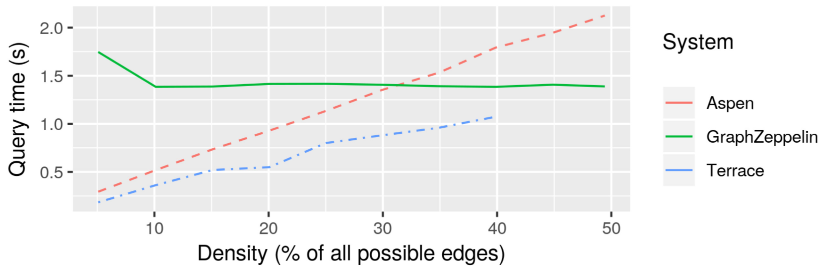
When the sketches fit in RAM, performance increases rapidly indicating that f can be quite small while providing a high ingestion rate. However, once memory requirements exceed main memory, f must be larger to offset disk IOs. To achieve an ingestion rate within 5% of peak performance on kron17, f as small as 0.01 is sufficient for entirely in RAM computation, while $f = .50$ is required when node sketches partially reside on disk.

6.6 Connectivity Queries Are Fast

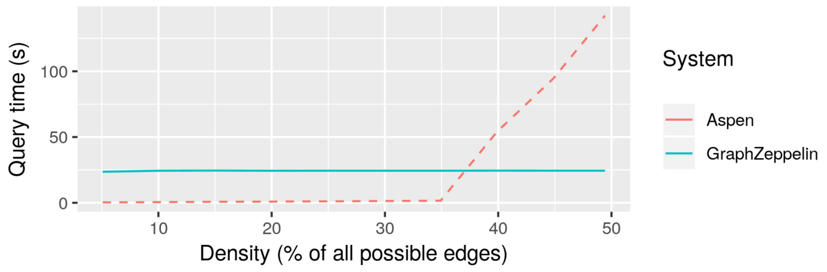
We show experimentally that GraphZeppelin gives comparable query performance to Aspen and Terrace on dense graphs when all systems' data structures fit in RAM. When their data structures reside on disk, GraphZeppelin answers queries more than five times faster than Aspen (and Terrace ingests too slowly to test).

GraphZeppelin's buffering strategies create a tradeoff between stream-ingestion rate and query latency. When GraphZeppelin receives a connectivity query, it must process remaining stream updates in its buffering system before computing connectivity using Boruvka's algorithm. Large buffers improve stream-ingestion rate (see Section 6.5), particularly when sketches are stored on disk, but this comes at the cost of increased query latency since these large buffers must be emptied. For the same reasons, small buffers improve query latency but may decrease the ingestion rate.

Figure 18(a) compares the query latency of GraphZeppelin, Aspen, and Terrace on the kron17 stream where connectivity queries are issued as graph density increases during the stream. In this experiment GraphZeppelin used small 400-byte leaf-only buffers, enough space for 100 stream updates. When the graph is sparser, both Aspen and Terrace answer queries more quickly than GraphZeppelin. As the stream progresses and the graph becomes denser, GraphZeppelin's query time stays constant while Aspen's and Terrace's increase. By 30% density, GraphZeppelin is faster



(a) In-memory query times.



(b) On-disk query times.

Fig. 18. GraphZeppelin query performance is comparable to or better than Aspen and Terrace for dense graphs.

than Aspen, though Terrace remains the fastest. Even with GraphZeppelin's small buffer size its ingestion rate was 3.95 million updates per second, three as fast as Aspen and almost six times faster than Terrace (until Terrace exceeds RAM size and slows drastically).

Figure 18(b) compares the query latency of GraphZeppelin and Aspen when RAM is limited to 12 GiB, forcing both systems to store part of their data structures on disk. Terrace ingests too slowly given only 12 GiB of RAM to be included in the experiment. In this experiment, GraphZeppelin used 8.3 KB leaf-only buffers (one-tenth of sketch size). GraphZeppelin takes 24 seconds to perform queries regardless of graph density. Aspen's queries are fast until the graph is too dense to fit in RAM; its last query takes 142 seconds, five times slower than GraphZeppelin. Notably, GraphZeppelin maintains an ingestion rate of 4.15 million updates per second, 46 times faster than Aspen. Both systems spend the majority of time on insertions, where GraphZeppelin's advantages come through.

7 RELATED WORK

Graph Sketching Systems. A practical method for using linear-sketching algorithms for connected-components computation was presented in Tench et al. [75]. They show how techniques from the AGM connectivity sketch [3] can used to develop a connected-components algorithm that is simultaneously space-optimal in the dynamic semi-streaming model and I/O-efficient in the external-memory model. They also built a graph-stream-processing system which is compared with state-of-the-art graph streaming systems [23, 66].

The present article serves as the journal version for Tench et al. [75], but it also contains follow-up work. Specifically, the evaluation in Tench et al. focus primarily on measuring the graph-stream-ingestion rate, whereas this article's expanded evaluation also focuses heavily on measuring (a) query latency and (b) the sensitivity of ingestion and query performance as a function of graph density.

In particular, this article contains new experiments in order to (1) compare the query latency of different ℓ_0-sketching algorithms, and to (2) evaluate the ingestion rate and query latency of GraphZeppelin and other streaming-graph processing systems as a function of graph density. This first class of experiments demonstrates how using the CubeSketch sketching algorithm (proposed in [75]) decreases GraphZeppelin's query latency by two orders of magnitude. The second class of experiments demonstrates that unlike existing graph-processing systems, GraphZeppelin's performance, remarkably, depends on the number of nodes, but not the density of edges.

Thus, this article shows that, contrary to conventional wisdom, computing on massive and dynamic graphs is possible even when these graphs are not sparse.

Graph Streaming Systems. Existing graph stream processing systems are designed primarily to handle updates in batches consisting entirely of insertions or entirely of deletions. Streaming systems that process updates in batches are generally divided into two categories. The first (which includes Terrace) consists of those systems which finish ingestion prior to beginning queries and finish queries prior to accepting any additional edges [7, 15, 24, 60, 66, 73, 74]. The second (which includes Aspen) allows updates to be applied asynchronously by periodically taking "snapshots" of the graph during ingestion to be used in conducting queries [17, 23, 35, 36, 54].

The batching employed in these systems limits the granularity at which insertions and deletions may be interspersed. In contrast, GraphZeppelin allows for insertions and deletions to be arbitrarily interspersed without sacrificing query correctness.

External Memory Systems. There is a rich literature of graph processing systems process static graphs in external memory. Some such systems store the entire graph out-of-core [31, 45, 53, 84, 86], and others are semi-external memory systems that maintain only the vertex-set in RAM [4, 52, 71, 83, 85]. Some systems provide (at least theoretical) design extensions to handle queries on graphs with insert-only updates [16, 45, 79, 80, 84], but to the best of our knowledge GraphZeppelin is the first to leverage external-memory effectively in the streaming model of insertions *and* deletions.

Practical Sketching Systems. While linear sketching was first implemented in a graph processing system in Tench et al. [75], linear sketching implementations for purposes other than graph processing have been widely studied. Some examples include sketches for recovering frequent items [22, 55] and estimating the cardinality of sets of items [11, 59]. Of particular note are sketches which realize the Johnson–Lindenstrauss lemma [37], which have found wide use in applications such as SDD system solvers and spectral sparsifiers [42, 43].

8 CONCLUSION

GraphZeppelin computes the connected components of graph streams using space asymptotically smaller than an explicit representation of the graph. It is based on CubeSketch, a new ℓ_0-sketching data structure that outperforms the state-of-the-art on graph-streaming workloads. This new sketching technique allows GraphZeppelin to process larger, denser graphs than existing graph-streaming systems given a fixed RAM budget and to ingest these graph streams more quickly. Even when GraphZeppelin's sketch data structures are too large to fit in RAM, its work-buffering strategies allow it to process graph streams on SSD. GraphZeppelin is simultaneously a space-optimal graph semi-streaming algorithm and an I/O-efficient external-memory algorithm.

The small space complexity of GraphZeppelin's linear sketch is optimized for large, dense graphs, unlike prior graph-processing systems, which often focus on sparse graphs. Thus, GraphZeppelin demonstrates that computational questions on graphs once thought intractably large and dense are now within reach.

Currently large, dense graphs are studied rarely and at great cost on large high-performance clusters [19]. Finding more applications that require processing large, dense graphs is an exciting direction for future work. Since GraphZeppelin's sketches can be updated independently (Section 5.1), we believe that they can be partitioned throughout a distributed cluster without sacrificing stream ingestion rate.

GraphZeppelin illustrates that additional algorithmic improvements help make graph semi-steaming algorithms into a powerful engineering tool by reducing the update-time complexity and allowing sketches to be stored efficiently on SSD. These techniques may generalize to other graph-analytics problems.

The AGM connected components sketch is a crucial subroutine for other graph sketching algorithms including approximate and exact MST, k-edge connectivity and vertex connectivity, minimum cut, spectral sparsifiers, and more. However, unchanged these algorithms suffer from similar issues to the original connectivity sketch—they are larger than RAM and have computationally expensive update procedures. Developing implementable and I/O efficient versions of these algorithms is an exciting direction for future work.

REFERENCES

[1] James Abello, Adam L. Buchsbaum, and Jeffery R. Westbrook. 2002. A functional approach to external graph algorithms. *Algorithmica* 32, 3 (2002), 437–458.

[2] Kook Ahn, Sudipto Guha, and Andrew McGregor. 2012. Graph Sketches: Sparsification, spanners, and subgraphs. *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* 1 (2012), 5–14. DOI: https://doi.org/10.1145/2213556.2213560

[3] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. 2012. Analyzing graph structure via linear measurements. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'12)*. 459–467.

[4] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2018. Clip: A disk I/O focused parallel out-of-core graph processing system. *IEEE Transactions on Parallel and Distributed Systems* 30, 1 (2018), 45–62.

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

[6] Stefano Allegretti, Federico Bolelli, Michele Cancilla, and Costantino Grana. 2018. Optimizing GPU-based connected components labeling algorithms. In *Proceedings of the 2018 IEEE International Conference on Image Processing, Applications and Systems (IPAS'18)*. 175–180. DOI: https://doi.org/10.1109/IPAS.2018.8708900

[7] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proceedings of the VLDB Endowment* 11, 6 (2018), 691–704. DOI: https://doi.org/10.14778/3184470.3184473

[8] J. Ang, Brian W. Barrett, Kyle B. Wheeler, and Richard C. Murphy. 2010. Introducing the graph 500. In *Proceedings of the Cray User Group (CUG)*. 45–74.

[9] Lars Arge. 1995. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the Workshop on Algorithms and Data Structures (WADS'95)*. 334–345.

[10] Tanya Y. Berger-Wolf and Jared Saia. 2006. 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)*. 523–528. DOI: https://doi.org/10.1145/1150402.1150462

[11] Kevin Beyer, Rainer Gemulla, Peter J. Haas, Berthold Reinwald, and Yannis Sismanis. 2009. Distinct-value synopses for multiset operations. *Communications of the ACM* 52, 10 (2009), 87–95.

[12] Ilaria Bordino and Debora Donato. 2009. Dynamic characterization of a large Web graph. In *Proceedings of the Web Science (WebSci'09)*.

[13] Gerth Stølting Brodal, Rolf Fagerberg, David Hammer, Ulrich Meyer, Manuel Penschuck, and Hung Tran. 2021. An experimental study of external memory algorithms for connected components. In *Proceedings of the19th International Symposium on Experimental Algorithms (SEA'21)*. 23:1–23:23.

[14] Libor Buš and Pavel Tvrdik. 2001. A parallel algorithm for connected components on distributed memory machines. In *Proceedings of the European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting (EuroMPI'01)*. 280–287.

[15] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. 2018. Hornet: An efficient data structure for dynamic sparse graphs and matrices on GPUs. In *Proceedings of the 2018 IEEE High Performance Extreme Computing Conference (HPEC'18)*. 1–7.

[16] Jiefeng Cheng, Qin Liu, Zhenguo Li, Wei Fan, John C. S. Lui, and Cheng He. 2015. VENUS: Vertex-centric streamlined graph computation on a single PC. In *Proceedings of the IEEE 31st International Conference on Data Engineering (ICDE'15)*. 1131–1142.

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

[18] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. 1995. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'95)*. 139–149.

[19] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1804–1815. DOI: https://doi.org/10.14778/2824032.2824077

[20] Yann Collet. 2016. xxHash-Extremely fast non-cryptographic hash algorithm. Retrieved from https://github.com/Cyan4973/xxHash. Accessed March 1, 2023.

[21] Graham Cormode and Donatella Firmani. 2014. A unifying framework for ℓ_0-sampling algorithms. *Distributed and Parallel Databases* 32 (2014), 315–335. DOI: https://doi.org/10.1007/s10619-013-7131-9

[22] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding frequent items in data streams. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1530–1541.

[23] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 918–934. Retrieved from https://par.nsf.gov/biblio/10137103

[24] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. 2012. Stinger: High performance data structure for streaming graphs. In *Proceedings of the 2012 IEEE Conference on High Performance Extreme Computing (HPEC'12)*. 1–5.

[25] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'96)*. 226–231.

[26] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective community search for large attributed graphs. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1233–1244. DOI: https://doi.org/10.14778/2994509.2994538

[27] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. 2005. On graph problems in a semi-streaming model. *Theoretical Computer Science* 348, 2 (2005), 207–216. DOI: https://doi.org/10.1016/j.tcs.2005.09.013

[28] Evangelos Georganas, Rob Egan, Steven Hofmeyr, Eugene Goltsman, Bill Arndt, Andrew Tritt, Aydin Buluç, Leonid Oliker, and Katherine Yelick. 2018. Extreme scale de novo metagenome assembly. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'18)*. 13 pages.

[29] John Greiner. 1994. A comparison of parallel algorithms for connected components. In *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'94)*. 16–25.

[30] Sudipto Guha, Andrew McGregor, and David Tench. 2015. Vertex and hyperedge connectivity in dynamic graph streams. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'15)*. 241–247.

[31] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'13)*. 77–85.

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

[33] Lifeng He, Xiwei Ren, Qihang Gao, Xiao Zhao, Bin Yao, and Yuyan Chao. 2017. The connected-component labeling problem: A review of state-of-the-art algorithms. *Pattern Recognition* 70 (2017), 25–43. DOI: https://doi.org/10.1016/j.patcog.2017.04.018

[34] M. Moftah Hossam, Aboul Ella Hassanien, and Mohamoud Shoman. 2010. 3D brain tumor segmentation scheme using K-mean clustering and connected component labeling algorithms. In *Proceedings of the10th International Conference on Intelligent Systems Design and Applications (ISDA'10)*. 320–324. DOI: https://doi.org/10.1109/ISDA.2010.5687244

[35] Anand Iyer, Li Erran Li, and Ion Stoica. 2015. CellIQ: Real-time cellular network analytics at scale. In *Proceedings of the12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. 309–322.

[36] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. In *Proceedings of the 4th International Workshop on Graph Data Management Experiences and Systems (GRADES'16)*. 1–6.

[37] William Johnson and J. Lindenstrauss. 1982. Extensions of lipschitz mappings into a hilbert space. *Conference in Modern Analysis and Probability* 26 (1982), 189–206.

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

[39] U. Kang and Christos Faloutsos. 2011. Beyond 'caveman communities': Hubs and spokes for graph compression and mining. In *Proceedings of the IEEE International Conference on Data Mining (ICDM'11)*. 300–309. DOI: https://doi.org/10.1109/ICDM.2011.26

[40] U. Kang, Mary McGlohon, Leman Akoglu, and Christos Faloutsos. 2010. Patterns on the connected components of terabyte-scale graphs. In *Proceedings of the IEEE International Conference on Data Mining (ICDM'10)*. 875–880. DOI: https://doi.org/10.1109/ICDM.2010.121

[41] Michael Korn, Daniel Sanders, and Josef Pauli. 2017. Moving object detection by connected component labeling of point cloud registration outliers on the GPU. In *Proceedings of the International Joint Conference on Computer Vision, Imaging, and Computer Graphics Theory and Applications (VISIGRAPP'17)*. 499–508.

[42] Ioannis Koutis, Alex Levin, and Richard Peng. 2015. Faster spectral sparsification and numerical algorithms for SDD matrices. *ACM Transactions on Algorithms* 12, 2 (2015), 1–16.

[43] Ioannis Koutis, Gary L. Miller, and Richard Peng. 2011. A nearly-m log n time solver for sdd linear systems. In *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. IEEE, 590–598.

[44] Arvind Krishnamurthy, Steven Lumetta, David E. Culler, and Katherine Yelick. 1997. Connected components on distributed memory machines. *Third DIMACS Implementation Challenge* 30 (1997), 1–21.

[45] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 31–46.

[46] Wookey Lee, James J. Lee, and Jinho Kim. 2014. Social network community detection using strongly connected components. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'14)*. 596–604.

[47] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. 2007. The dynamics of viral marketing. *ACM Transactions on the Web* 1, 1 (2007), 5.

[48] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. Retrieved from http://snap.stanford.edu/data. Accessed March 1, 2023.

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

[50] Yongsub Lim, Won-Jo Lee, Ho-Jin Choi, and U. Kang. 2015. Discovering large subsets with high quality partitions in real world graphs. In *Proceedings of the 2015 International Conference on Big Data and Smart Computing (BIGCOMP'15)*. 186–193.

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

[52] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-grained I/O management for graph computing. In *Proceedings of the15th USENIX Conference on File and Storage Technologies (FAST'17)*. 285–300.

[53] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*. 527–543.

[54] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. Llama: Efficient graph analytics using large multiversioned arrays. In *Proceedings of the 2015 IEEE 31st International Conference on Data Engineering (ICDE'15)*. 363–374.

[55] Nishad Manerikar and Themis Palpanas. 2009. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data and Knowledge Engineering* 68, 4 (2009), 415–430.

[56] Julian J. McAuley and Jure Leskovec. 2012. Learning to discover social circles in ego networks.. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS'12)*. 548–56.

[57] Andrew McGregor, David Tench, Sofya Vorotnikova, and Hoa T. Vu. 2015. Densest subgraph in dynamic graph streams. In *Proceedings of the Mathematical Foundations of Computer Science (MFCS'15)*. 472–482.

[58] Duccio Medini, Antonello Covacci, and Claudio Donati. 2006. Protein homology network families reveal step-wise diversification of Type III and Type IV secretion systems. *PLoS Computational Biology* 2, 12 (2006), 173.

[59] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2008. Why go logarithmic if we can go linear? Towards effective distinct counting of search traffic. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*. 618–629.

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

[61] S. Muthukrishnan. 2005. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science* 1, 2 (2005), 117–236. DOI: https://doi.org/10.1561/0400000002

[62] Jelani Nelson and Huacheng Yu. 2019. Optimal lower bounds for distributed and streaming spanning forest computation. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'19)*. 1844–1860.

[63] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. 2001. Otakar BorůVka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Math* 233, 1–3 (2001), 3–36. DOI: https://doi.org/10.1016/S0012-365X(00)00224-7

[64] Sergey Nurk, Dmitry Meleshko, Anton Korobeynikov, and Pavel Pevzner. 2017. MetaSPAdes: A new versatile metagenomic assembler. *Genome Research* 27 (2017), 824–834. DOI: https://doi.org/10.1101/gr.213959.116

[65] OpenMP Architecture Review Board 2018. *OpenMP Application Programming Interface* (5th ed.). OpenMP Architecture Review Board.

[66] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluç. 2021. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD'21)*. 1372–1385.

[67] Md. Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. 2012. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. 1–11. DOI: https://doi.org/10.1109/SC.2012.9

[68] Alex Pothen and Chin-Ju Fan. 1990. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software* 16, 4 (1990), 303–324. DOI: https://doi.org/10.1145/98267.98287

[69] Matei Ripeanu and Ian Foster. 2002. Mapping the gnutella network: Macroscopic properties of large-scale peer-to-peer systems. *Peer-to-Peer Systems*, Peter Druschel, Frans Kaashoek, and Antony Rowstron (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 85–93.

[70] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI'15)*. 4292–4293. Retrieved from https://networkrepository.com

[71] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. 472–488.

[72] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment* 11, 4 (2017), 420–431. DOI: https://doi.org/10.1145/3186728.3164139

[73] Dipanjan Sengupta and Shuaiwen Leon Song. 2017. EvoGraph: On-the-fly efficient mining of evolving graphs on GPU. In *Proceedings of the International Supercomputing Conference (ISC'17)*. 97–119.

[74] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. 2016. Graphin: An online high performance incremental graph processing framework. In *Proceedings of the European Conference on Parallel Processing (Euro-Par'16)*. 319–333.

[75] David Tench, Evan West, Victor Zhang, Michael A. Bender, Abiyaz Chowdhury, J. Ahmed Dellas, Martin Farach-Colton, Tyler Seip, and Kenny Zhang. 2022. GraphZeppelin: Storage-friendly sketching for connected components on dynamic graph streams. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD'22)*. Association for Computing Machinery, New York, NY, USA, 325–339. DOI: https://doi.org/10.1145/3514221.3526146

[76] Heidi Thornquist, Eric Keiter, Robert Hoekstra, David Day, and Erik Boman. 2009. A parallel preconditioning strategy for efficient transistor-level circuit simulation. In *Proceedings of the 2009 International Conference on Computer-Aided Design (ICCAD'09)*. 410–417. DOI: https://doi.org/10.1145/1687399.1687477

[77] Stijn Marinus Van Dongen. 2000. *Graph Clustering by Flow Simulation*. Ph.D. Dissertation. University Utrecht.

[78] Jeffrey Scott Vitter. 2001. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys* 33, 2 (2001), 209–271.

[79] Keval Vora. 2019. LUMOS: Dependency-driven disk-based graph processing. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*. 429–442.

[80] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*. 507–522.

[81] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Efficient structural graph clustering: An index-based approach. *The VLDB Journal* 28, 3 (2019), 377–399. DOI: https://doi.org/10.1007/s00778-019-00541-4

[82] Min Wu, Xiaoli li, Chee-Keong Kwoh, and See-Kiong Ng. 2009. A core-attachment based method to detect protein complexes in PPI networks. *BMC Bioinformatics* 10 (2009), 169. DOI: https://doi.org/10.1186/1471-2105-10-169

[83] Pingpeng Yuan, Changfeng Xie, Ling Liu, and Hai Jin. 2016. PathGraph: A path centric graph processing system. *IEEE Transactions on Parallel and Distributed Systems* 27, 10 (2016), 2998–3012.
[84] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A novel abstraction-based out-of-core graph processing system. *ACM SIGPLAN Notices* 53, 2 (2018), 608–621.
[85] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *Proceedings of the13th USENIX Conference on File and Storage Technologies (FAST'15).* 45–58.
[86] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15).* 375–386.

Received 16 March 2023; revised 11 October 2023; accepted 24 January 2024