Graph Stream Sketch: Summarizing Graph Streams with High Speed and Accuracy

Xiangyang Gou, Lei Zou, Chenxingyu Zhao and Tong Yang

Abstract—A graph stream is a continuous sequence of data items, in which each item indicates an edge, including its two endpoints and edge weight. It forms a dynamic graph that changes with every item. Graph streams play important roles in cyber security, social networks, cloud troubleshooting systems and more. Due to the vast volume and high update speed of graph streams, traditional data structures for graph storage such as the adjacency matrix and the adjacency list are no longer sufficient. However, prior art of graph stream summarization either supports limited kinds of queries or suffers from poor accuracy of query results. In this paper, we propose a novel *G*raph *S*tream *S*ketch (GSS for short) to summarize the graph streams, which has linear space cost $O(|E|)$ (E is the edge set of the graph) and high update speed, and supports most kinds of queries over graph streams with controllable errors. Experimental results show that our solution is up to 142 times faster than the adjacency list when processing updates in graph streams, and its memory consumption is as small as 30% of the adjacency list. Though error is introduced as a trade off in our solution, both theoretical analysis and experiment results confirm that such error is small and controllable. The relative error is below 10^{-2} in edge weight query, and the precision is above 90% is 1-hop precursor/successor queries.

Index Terms—graph, data stream, sketch, approximate query

1 Introduction

In the era of big data, data streams propose new challenges to existing systems. Furthermore, the traditional data stream is modeled as a sequence of *isolated* items, and the connections between these items are rarely considered. However, in many data stream applications, the connections often play important roles in data analysis, such as finding malicious attacks in network traffic data, mining news spreading paths among social networks. In these cases the data are organized as *graph streams*. A graph stream is an unbounded sequence of items, in which each item is denoted as $(\overrightarrow{s,d};w;t)$, where $\overrightarrow{s,d}$ represents an edge from nodes s to d, w is the edge weight and t is the timestamp. These data items together form a *streaming graph* that changes continuously. Note that, for ease of presentation, we use the terms "graph stream" and "streaming graph" interchangeably in this paper. Below we discuss an example to demonstrate the usefulness of streaming graphs.

Use case 1: *Network traffic.* The network traffic can be seen as a large streaming graph, where each edge indicates the communication between two IP addresses. With the arrival of packets in the network, the network traffic graph changes rapidly and constantly. In the network traffic graph, various kinds of queries are needed, like performing node queries to find malicious attackers, or subgraph queries to locate certain topology structures in the dynamic network.

Use case 2: *Social networks.* In a social network, interactions among users form a streaming graph. Edges between different nodes are weighted by the frequencies of interactions. In such a graph, queries like finding the potential friends of a user and tracking the spreading path of a piece of news are often needed.

Many real-world streaming graphs have large sizes and high throughput. For example, in large ISP or data centers [1], there could be millions of packets every second. The large volume and high dynamicity make it hard to store the whole graph stream efficiently with traditional data structures, such as adjacency lists or adjacency matrices. Considering the above graph streaming applications, there are two requirements for designing a new data structure : (1) linear space cost; and (2) high update speed. There have been works for graph summarization based on grouping nodes or edges with similar neighborhood, like [2], [3], but they either do not support updates or have a low update speed. Approximate data structures for traditional data streams, like CM sketch [4] and other sketches [5], [6] can also be considered, but they support limited query types. In recent years, data structures for approximate graph stream summarization with high speed are also proposed, like TCM [7] and gMatrix [8]. But their accuracy is quite low. More related work is discussed in Section 2.

In this paper, we design a novel data structure–Graph *S*tream *S*ketch (GSS for short) to support most kinds of queries over streaming graphs with controllable errors in query results. Both theoretical analysis and experiment results show that the accuracy of our method outperforms state-of-the-arts by orders of magnitude.

- *Xiangyang Gou is with Peking University, China. E-mail: gxy1995@pku.edu.cn*
- *Corresponding author: Lei Zou is with Peking University, Beijing Institute of Big Data Research and National Engineering Laboratory for Big Data Analysis Technology and Application (PKU), China. E-mail:zoulei@pku.edu.cn*
- *Chenxingyu Zhao is with Peking University, China. E-mail: dkzcxy@pku.edu.cn*
- *Tong Yang is with Peking University, China. E-mail: yangtongemail@gmail.com*

1.1 Our Solution

In this paper, we propose an approximate data structure (GSS) for graph streams with linear memory usage, high update speed and high accuracy. Moreover, GSS supports most kinds of graph queries and algorithms.

Like TCM, GSS uses a hash function $H(\cdot)$ to compress a streaming graph G into a smaller one G_h, named a *graph sketch*. Each node v in G is hashed to $H(v)$. Nodes in G with the same hash value are condensed into one node in G_h, and the edges connected to them are also aggregated. An example of the graph stream G and the graph sketch G_h can be referred in Figures 1 and 2, respectively. The compression ratio can be controlled by the hash range of $H(\cdot)$ (denoted as M). Obviously, the higher the compression ratio is, the lower the accuracy is.

TCM uses the adjacency matrix to store the graph sketch. However, the adjacency matrix is far from memory efficient when storing large sparse graphs. Its memory usage is $O(|V|^2)$, where $|V|$ is the number of nodes. In order to control the memory usage, TCM has to heavily compress the streaming graph, resulting into low accuracy.

In GSS, we design a novel data structure to store the graph sketch, which combines fingerprints and hash addresses to distinguish nodes and edges. Compared to the adjacency matrix, it has higher memory efficiency and can store a larger graph sketch with the same space. Meanwhile, it also achieves high update speed. We further propose a technique called *square hashing*, which makes the data structure more compact and improves both space and time efficiency.

Note that GSS is designed to support various kinds of queries upon the streaming graph, thus we propose three query primitives based on GSS. They are edge query, 1-hop successor query and 1-hop precursor query. In Section 7, we propose several variants which improve the efficiency of these query primitives, especially the successor query and the precursor query.

To summarize, we made the following contributions:

1) We propose GSS, a novel data structure for graph stream summarization. It has small memory usage, high update speed, and supports most kinds of queries over streaming graphs.
2) We propose a technique called square hashing. It helps to compact the data structure of GSS, which improves update speed and reduces memory cost.
3) We define three graph query primitives and give details about how GSS supports them. Almost all algorithms for graphs can be implemented with these primitives. In order to further improve these query primitives, especially the successor query and the precursor query, we propose several improved versions of GSS.
4) We conduct theoretical analysis and extensive experiments to evaluate the performance of GSS, which show that our method outperforms state-of-the-art in terms of query accuracy and system throughput.

2 Related Work

Graph summarization and graph sketches have been investigated for years. They can be divided into three kinds:

The first kind summarizes the graph by grouping nodes and edges with similar neighborhood. For example, Fan *et.al.* [2] propose query-specific functions to group equivalence nodes, so that the compressed graph can answer specific queries without loss. Raghavan *et.al.* [9] propose a 2-level compressed representation of web graphs based on grouping small sets of web pages. Riondato *et.al.* [10] build connection between graph summarization and geometric clustering problems, and propose a lossy group-based graph summarization algorithm with accuracy guarantee. However, most of these works do not support dynamic graphs. Though a small part of algorithms in this kind support incremental summarizing, they still have a low speed due to the high cost of discovering similar nodes or edges. In Section 8.6, we compare the state-of-the-art incremental graph summarization method, MoSSo [3] with our work, and the result shows that it is up to 10^3 times slower.

The second kind summarizes the graph by selectively extracting edges and nodes. These algorithms only keep essential data to provide approximations of certain graph metrics, like sparsifiers for edge cut approximation and spanners for node distance approximation. For example, Peleg *et.al.* [11] build a k-spanner where the estimated node distance is within k times of the true value. And Spielman *et.al.* [12] build sparsifiers by sampling edges according to their effective resistance. Recent work has extended this kind to graph streams, like estimating maximum matching size [13], building sparsifiers and spanners [14], [15] and maintaining dense subgraphs [16] in graph streams. However, these algorithms can only answer typical kinds of queries, as large fraction of graph data is lost.

The third kind encodes the graph in bit level, like reordering edges [17] and bipartite minimum logarithmic arrangement [18], but they do not support graph updates.

More graph summarization algorithms can be referred in [19]. Besta *et.al.* [20] also propose a programming model that can combine different graph summarization methods.

There are some other works which aim to support certain kinds of continuous queries like triangle counting or subgraph matching [21]–[23], and systems which aim to provide high query performance while supporting graph updates, like [24]–[28]. They are built upon traditional graph storage structures like adjacency lists, and focus on query strategies. On the other hand, we aim to design a new graph storage structure which is more suitable for high-throughput graph streams. Therefore, we position our work as a competitor to the adjacency list rather than these works. As will be discussed in Section 8.6, we experimentally compare our algorithm GSS with the adjacency list. The results show that the memory usage of GSS is only $30\% \sim 50\%$ of the adjacency list, and the update speed of GSS is up to 142 times higher than the adjacency list. This confirms the superiority of GSS in storing high-throughput, large-volume graph streams. More related work about systems and query algorithms upon streaming graphs can be referred in [29]–[31].

Multiple variants of the graph stream models are also proposed. For example, semi-streaming model [32] allows

algorithms to process the graph data in multiple passes. This model suits the situation where large static graphs are stored on the disk. Algorithms can scan the graph multiple times from the disk, but cannot store it in memory. On the other hand, we define graph streams as sequences of edges arriving from data sources like internet, and we can only process them in one scan. Other variants include W-stream model [33] which allows stream manipulations across passes and stream-sort model [34] which allows stream sorting passes.

Data stream summarization has been another hot topic for years. Related work usually uses hash-based method to build compact data structures like counter arrays or bit arrays to summarize the data stream and provide approximate support to certain kinds of queries, like the CM sketch [4], the CU sketch [35] and so on [5], [6], [36].

In our paper, we follow the idea of applying techniques of data stream summarization to graph streams. We define the graph stream summarization problem as designing a data structure with linear memory usage, high update speed, and provides graph query primitives to support various kinds of graph queries. In this scenario, TCM [7] is the state-of-the-art for graph stream summarization. It uses a hash function $H(\cdot)$ to compress the streaming graph $G = (V, E)$ into a smaller graph sketch G_h. For each node v in G, TCM maps it to node $H(v)$ in G_h. For each edge $e = \overrightarrow{s, d}$ in G, TCM maps it to edge $\overrightarrow{H(s), H(d)}$ in G_h. The weight of an edge in G_h is an aggregation of the weight of all edges mapped to it. Then TCM uses an adjacency matrix to represent the graph sketch. When the memory is sufficient, we can also build multiple sketches with different hash functions, and report the most accurate value in queries.

If we represent the size of the value range of $H(\cdot)$ with M, we need to build an $M \times M$ adjacency matrix. In order to satisfy the demand on memory usage, the size of the matrix, $M \times M$ has to be within $O(|E|)$, which means $M \ll |V|$ for a sparse streaming graph. This means the graph sketch is much smaller than G, leading to heavy hash collisions and poor query accuracy in TCM. Following works include [8], [37], [38] which extend TCM to labeled graphs or support heavy hitter queries, but the problem of poor accuracy is still not solved.

3 Problem Definition

Definition 1. **Graph Stream**: A graph stream is an unbounded time evolving sequence of items $S = \{e_1, e_2, e_3 ..... e_n\}$, where each item $e_i = (\overrightarrow{s, d}; t; w)$ indicates a directed edge [1] from node s to node d, with wight w. The timepoint t_i is also referred as the timestamp of e_i. Thus, the edge streaming sequence S forms a directed graph $G = (V, E)$ that changes with the arrival of every item e_i, where V and E denote the set of nodes and the set of edges in the graph, respectively. We call G a *streaming graph* for convenience.

In a graph stream S, an edge $\overrightarrow{s, d}$ may appear multiple times with different timestamps. The weight of such an edge in the streaming graph G is SUM of weight of all these occurrences. In the majority of the paper, we suppose that the weight of each item, w, is a positive number. It means the stream only inserts edges but does not remove them. This insertion-only model applies to scenarios like network monitoring (IP as nodes and communications as edges) and social network monitoring (user as nodes and interactions as edges). GSS can be used to summarize data arriving in such streams in a period like days or weeks. In Section 5.2.1, we will further extend GSS to streams with negative weight and edge deletions. We also discuss how to handle graph with edge labels in Appendix C of the supplementary materials.

1. The approach in this paper can be easily extended to handle undirected graphs.

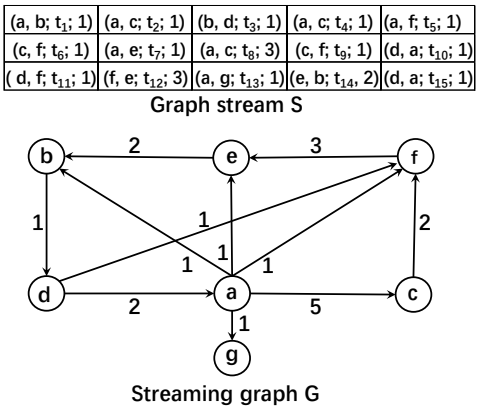


Fig. 1. An example of the graph stream

Example 1. An example of the graph stream, S, and the corresponding streaming graph G are both shown in Figure. 1. If an edge appears multiple times, the weight of these occurrences is added up as stated above.

In practice, G is usually a large, sparse and high speed dynamic graph. The large volume and high dynamicity make it hard to store graph streams using traditional data structures such as adjacency lists and adjacency matrices. The large space cost of $O(|V|^2)$ rules out the possibility of using the adjacency matrix to represent a large sparse graph. On the other hand, the adjacency list has $O(|E|)$ memory cost, which is acceptable. However, the time cost of updating is $O(|V|)$, as we have to search for the edge first, in order to determine if we should add a new edge, or just modify the weight of an existing edge. This is unacceptable due to the high speed of the graph stream.

The goal of our study is to design a *linear* space cost data structure with efficient update algorithm over high speed graph streams and support to various kinds of queries. To meet that goal, we allow some approximate query results but with small and controllable errors. However, prior solutions for graph summarization / data stream summarization / graph stream summarization suffer from different problems, like low update speed or even cannot update [2], [3], limited queries types [4], [39] or low accuracy [7], [8]. Therefore, in this paper, we design a novel graph stream summarization strategy.

Formally, we define our *graph stream summarization* problem as follows.

Definition 2. **Graph Stream Summarization**: Given a streaming graph $G = (V, E)$, the *graph stream summarization* problem is to design a compact data structure DS to represent the streaming graph, where the following conditions hold:

1) The space cost of DS is $O(|E|)$;
2) DS changes with each new arriving data item in the streaming graph and the time complexity of updating DS should be as small as possible;
3) DS supports various queries over the streaming graph G with small and controllable errors.

In order to support various kinds of graph queries, we define three graph query primitives.

Definition 3. **Graph Query Primitives**: Given a graph $G(V, E)$, the three graph query primitives are:

- **Edge Query**: given an edge $e = \overrightarrow{s, d}$, return its weight $w(e)$ if it exists in the graph and return -1 if not.
- **1-hop Successor Query**: given a node v, return a set of nodes that are 1-hop reachable from v, and return $\{-1\}$ if there is no such node;
- **1-hop Precursor Query**: given a node v, return a set of nodes that can reach node v in 1-hop, and return $\{-1\}$ if there is no such node.

With these primitives, we can retrieve all information in the streaming graph. Connection of nodes can be retrieved by 1-hop successor queries and 1-hop precursor queries. The weight of the edges can be retrieved by edge queries. Therefore, all kinds of queries and algorithms can be supported with these primitives.

The notations used in this paper are shown in Appendix A of the supplementary materials.

4 GSS: Basic Version

In this section, we describe a conceptually simple scheme to help to illustrate intuition and benefit of our approach. The full approach, presented in Section 5, is designed with more optimizations. To produce a graph stream summarization, we first design a graph sketch $G_h = (V_h, E_h)$ for the streaming graph G, which is a smaller graph generated by compressing G with hash functions.

We choose a hash function $H(\cdot)$ with value range $[0, M)$, and then G_h is generated as follows:

1) **Initialization**: Initially, $V_h = \varnothing$, and $E_h = \varnothing$.
2) **Edge Insertion**: For each edge $e = \overrightarrow{s, d}$ in E with weight w, we compute hash values $H(s)$ and $H(d)$. If either node $H(s)$ or $H(d)$ is not in V_h yet, we insert it into V_h. Then we set $H(e) = \overrightarrow{H(s), H(d)}$. If $H(e)$ is not in E_h, we insert $H(e)$ into E_h and set its weight $w(H(e)) = w$. If $H(e)$ is in E_h already, we add w to the weight.

G_h is empty at the beginning and expands with every data item in the graph stream. We can store $\langle H(v), v\rangle$ pairs with a hash table to make this mapping procedure reversible. This needs $O|V|$ additional memory, as $|V| \leqslant |E|$, the overall memory requirement is still within $O(|E|)$.

Example 2. A graph sketch G_h for the streaming graph G in Figure 1 is shown in Figure 2. The value range of the hash function $H(\cdot)$ is $[0, 32)$. In the example, nodes c and g are mapped to the same node with ID 5 in G_h. In G_h, the weight of edge $\overrightarrow{2, 5}$ is 6, which is the summary of the weight of edge $\overrightarrow{a, c}$ and edge $\overrightarrow{a, g}$ in G.

Node	a	b	c	d	e	f	g
H(v)	2	15	5	28	10	18	5

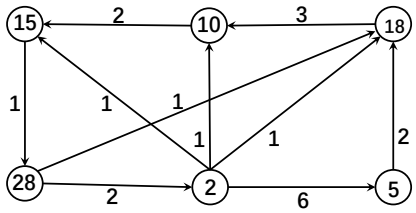


Fig. 2. An example of the graph sketch

Obviously, the size of the value range of the map function $H(\cdot)$, which we represent with M, will influence the size of the graph sketch. The generated graph sketch is always no larger than the original streaming graph, as the map function is a many-to-one map. The smaller M is, the smaller the graph sketch will become. Theoretically, when $M = \sigma \times |V|$, where $|V|$ is the number of nodes in the streaming graph, the generated graph sketch will have $(1 - e^{-\frac{1}{\sigma}})\sigma|V|$ nodes. We can control the size of the graph sketch by setting different M. We also transform the original node IDs, which may be long strings, to integers with $log(M)$ bits with the map function. It helps us to save space when storing the graph sketch.

However, it should be noted that when M becomes smaller, we will have a higher probability to get a wrong answer in queries, especially 1-hop successor query and 1-hop precursor query. In Appendix B of the supplementary materials, we demonstrate the theoretical results of the relationship between M and the accuracy of the query primitives with figures. The result shows that M has to be much larger than $|V|$ to get high accuracy in 1-hop successor / precursor queries.

TCM resorts to an adjacency matrix to represent G_h. In this case, the matrix width m equals to M, i.e, the value range of the map function. To keep the memory usage within $O(|E|)$ (Condition 1 in Definition 2), m must be less than $\sqrt{|E|}$, that means $m = M < \sqrt{|E|} \ll |V|$ for a sparse streaming graph. Large quantities of nodes will collide with each other, leading to low accuracy. Our theoretical analysis in Section 6.1 and experiments in Section 8 confirm this.

Node	a	b	c	d	e	f	g
H(v)	2	15	5	28	10	18	5
<h(v), f(v)>	<0, 2>	<1, 7>	<0, 5>	<3, 4>	<1, 2>	(2, 2)	(0, 5)

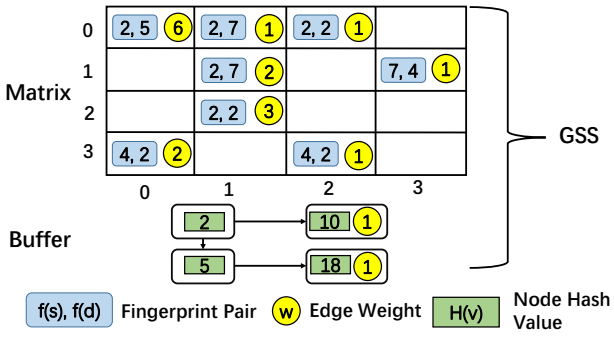


Fig. 3. An example of the basic version of data structure

Considering the above limitations, we design a novel data structure for *g*raph *s*tream *s*ummarization, called GSS.

Definition 4. **GSS**: Given a streaming graph $G = (V, E)$, we have a hash function $H(\cdot)$ with value range $[0, M)$ to map each node v in graph G to node $H(v)$ in graph sketch G_h. Then we use the following data structure to represent the graph sketch G_h:

1) GSS consists of a size $m \times m$ adjacency matrix X and an adjacency list buffer B for left-over edges.
2) For each node $H(v)$ in sketch graph G_h, we define an address $h(v)(0 \leqslant h(v) < m)$ and a fingerprint $f(v)(0 \leqslant f(v) < F)$ where $M = m \times F$ and $h(v) = \lfloor \frac{H(v)}{F} \rfloor$, $f(v) = H(v)\%F$.
3) Each edge $\overrightarrow{H(s), H(d)}$ in the graph sketch G_h is mapped to a bucket in the row $h(s)$, column $h(d)$ of the matrix X. We record $\langle f(s), f(d) \rangle$ and w in the corresponding bucket of the matrix, where w is the edge weight and $f(s)$, $f(d)$ are fingerprints of the two endpoints.
4) Adjacency list buffer B records all left-over edges in G_h, whose expected positions in the matrix X have been occupied by previous inserted edges.

When implementing a GSS for a graph stream, in order to satisfy the $O(|E|)$ memory cost requirement, we usually set $m = \alpha \times \sqrt{|E|}$, where α should be a constant approximate to 1. To achieve high accuracy, we set $M \gg |V|$. This can be achieved by setting a large F, in other words, using long fingerprints. When the memory is not sufficient, we can also set smaller M with smaller m and F, but this will decrease the accuracy.

Example 3. The basic version of GSS to store G_h in Figure 2 is shown in Figure 3. Here we set $F = 8$. The nodes in the original streaming graph and their corresponding $H(v)$, $h(v)$ and $f(v)$ are shown in the table. In this example, edge $\overrightarrow{2, 10}$ and edge $\overrightarrow{5, 18}$ in G_h are stored in the buffer because of collisions with other edges.

In GSS, we store edges with different source nodes in G_h in one row of the matrix, because the graph is sparse and each node is usually connected to very few edges. We can use fingerprints to distinguish them. It is similar in columns. This idea of combining addresses and fingerprints to distinguish different items is known as quotienting [40] and is widely used in hash-based structures. Fingerprints also help us to distinguish edges when they are mapped into the same bucket. This enables us to apply a map function with a much larger value range, and generate a much larger graph sketch with the same matrix size as TCM.

5 GSS: An Optimized Version

As we know, GSS has two parts: a size $m \times m$ matrix X and an adjacency list buffer B for left-over edges. Obviously, we only need $O(1)$ time to insert an edge into X, but linear time $O(|B|)$ if the edge must go to the buffer B, where $|B|$ represents the number of all left-over edges. Therefore, $|B|$ influences both the memory and the time cost. In this section, we design a solution, namely square hashing, to reduce the $|B|$. Then we further propose several improvements to increase the update speed.

5.1 Square Hashing

In the basic version, an edge is pushed into buffer B if its mapped position in the matrix X has been occupied. The most intuitive solution is to find another bucket for it. We further notice the highly skewed degree distribution in real-world graphs, in which node degrees usually follow the power-law distribution. In other words, a few nodes have very high degrees, while most nodes have small degrees. Consider a node v that has A out-going edges in the graph sketch G_h. For a $m \times m$ adjacency matrix X in GSS (see Definition 4), there are at least $A - m$ edges that should be inserted into buffer B, as these A edges must be mapped to the same row (in X) due to the same source vertex v. These high degree nodes lead to crowded rows and result in most of the left-over edges. On the other hand, many other rows are uncrowded. We have the same observation for columns of matrix X. *Is it possible to make use of unoccupied positions in uncrowded rows / columns?* It is the motivation of our first technique, called *square hashing*.

For each node with ID $H(v) = \langle h(v), f(v) \rangle$ in G_h, we compute a sequence of hash addresses $\{h_i(v)|1 \leqslant i \leqslant r\}, (0 \leqslant h_i(v) < m)$ for it. Edge $\overrightarrow{H(s), H(d)}$ is stored in the first empty bucket among the $r \times r$ buckets with addresses

$$\{\langle h_{i_s}(s), h_{i_d}(d) \rangle | (1 \leqslant i_s \leqslant r, 1 \leqslant i_d \leqslant r)\}$$

where $h_{i_s}(s)$ is the row index and $h_{i_d}(d)$ is the column index. We call these buckets *mapped buckets* for convenience. Note that we consider row-first layout when selecting the first empty bucket.

The following issue is how to generate a *good* hash address sequence $\{h_i(v)|1 \leq i \leq r\}$ for a vertex v. There are two requirements:

Independent: For two nodes v_1 and v_2, we use Pr to represent the probability that $\forall 1 \leq i \leq r, h_i(v_1) = h_i(v_2)$. Then we have $Pr = \prod_{i=1}^{r} Pr(h_i(v_1) = h_i(v_2))$. In other words, the randomness of each address in the sequence will not be influenced by others. This requirement will help to maximize the chance that an edge finds an empty bucket among the $r \times r$ mapped buckets.

Reversible: Given a bucket in row R and column C and the content in it, we are able to recover the representation of the edge e in the graph sketch G_h: $\overrightarrow{H(s), H(d)}$, where e is the edge in that bucket. This property is needed in the 1-hop successor query and the 1-hop precursor query. As in these queries, we need to check the potential buckets to see if they contain edges connected to the queried node v and retrieve the other end point in each qualified bucket.

To meet the above requirements, we propose to use *linear congruence method* [41] to generate a sequence of r random values $\{q_i(v)|1 \leqslant i \leqslant r\}$ with $f(v)$ as seeds. We call this sequence the linear congruential (LR) sequence for convenience. The linear congruence method is as following: select a timer a, small prime b and a module p, then

$$\begin{cases} q_1(v) = (a \times f(v) + b)\%p \\ q_i(v) = (a \times q_{i-1}(v) + b)\%p, (2 \leqslant i \leqslant r) \end{cases} \quad (1)$$

By choosing a, b and p carefully, we can make sure the cycle of the sequence we generate is much larger than r, and there will be no repetitive numbers in the sequence

[41]. Then we generate a sequence of hash addresses as following:

$$\{h_i(v)|h_i(v) = (h(v) + q_i(v))\%m, 1 \leqslant i \leqslant r\} \quad (2)$$

When storing edge $\overrightarrow{H(s), H(d)}$ in the matrix, besides storing the pair of fingerprints and the edge weight, we also store an index pair $\langle i_s, i_d\rangle$, supposing that the bucket has an address $\langle h_{i_s}(s), h_{i_d}(d)\rangle$. As the length of the sequence, r, is small, the length of each index will be less than 4 bits. Therefore storing such a pair will cost little.

Note that the hash sequence $\{q_i(v)|1 \leqslant i \leqslant r\}$ generated by the linear congruence method are both *independent* and *reversible*. The independence property has been proved in [41]. We show how to recover the original hash value $H(v)$ based on the $f(v)$, $h_i(v)$ and the index i as follows. First, we compute the LR sequence $\{q_i(v)\}$ with $f(v)$ following equation 1. Second, we use equation $(h(v) + q_i(v))\%m = h_i(v)$ to compute the original hash address $h(v)$. As $h(v) < m$, the equation has a unique solution. At last we use $H(v) = h(v) \times F + f(v)$ to compute $H(v)$. Given a bucket in the matrix, the fingerprint pair $\langle f(s), f(d)\rangle$ and the index pair $\langle i_s, i_d\rangle$ are all stored in it, and we have $h_{i_s}(s) = R$, $h_{i_d}(d) = C$, where R and C are the row index and the column index of the bucket in the matrix, respectively. Therefore we can retrieve both $H(s)$ and $H(d)$ as above.

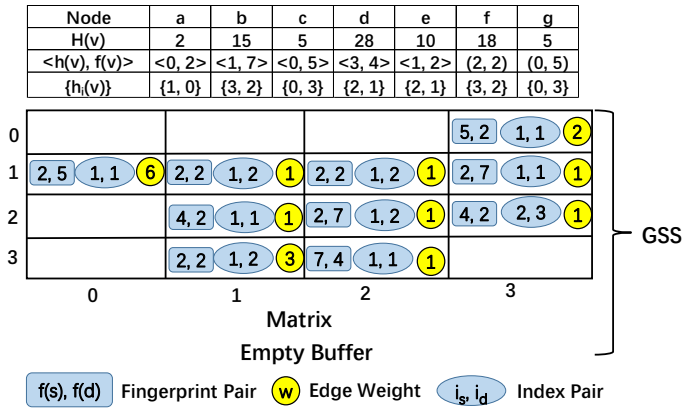


Fig. 4. An example of the modified version of data structure

Example 4. An example of the modified version is shown in Figure. 4. In the matrix we store G_h in Figure. 2, which is a graph sketch of G in Figure 1. In this example we set $F = 8$, $m = 4$, $r = 2$, and the equation in the linger congruence method is

$$\begin{cases} q_1(v) = (5 \times f(v) + 3)\%8 \\ q_i(v) = (5 \times q_{i-1}(v) + 3)\%8, (2 \leqslant i \leqslant r) \end{cases} \quad (3)$$

Compared to the basic version, in the modified version all edges are stored in the matrix, and the number of memory accesses we need to find an edge in the matrix is within $2^2 = 4$. In fact in the example we only need one memory access to find most edges, and 2 for a few ones.

We illustrate four basic operators in GSS as follows.

Edge Updating: When a new item $(\overrightarrow{s, d}; t; w)$ comes in the graph stream S, we map it to edge $\overrightarrow{H(s), H(d)}$ in the graph sketch G_h with weight w. Then we compute two hash address sequences $\{h_i(s)\}$ and $\{h_i(d)\}$ and check the r^2 mapped buckets with addresses $\{\langle h_i(s), h_j(d)\rangle|1 \leqslant i \leqslant r, 1 \leqslant j \leqslant r\}$ one by one. For a bucket in row $h_{i_s}(s)$ and column $h_{i_d}(d)$, if it is empty, we store the fingerprint pair $\langle f(s), f(d)\rangle$, the index pair $\langle i_s, i_d\rangle$ and weight w in it, and end the procedure. If it is not empty, we check the fingerprint pair $\langle f(s'), f(d')\rangle$ and the index pair $\langle i'_s, i'_d\rangle$ stored in the bucket. If the fingerprint pair and the index pair are both equal to the corresponding pairs of the inserted edge $\overrightarrow{H(s), H(d)}$, we add w to the weight in it, and end the procedure. Otherwise it means this bucket has been occupied by another edge, and we consider other hash addresses following the hash sequence. If all r^2 buckets have been occupied, we store edge $\overrightarrow{H(s), H(d)}$ with weight w in the buffer B.

Graph Query Primitives: The three graph query primitives are supported as follows:

Edge Query: When querying an edge $e = \overrightarrow{s, d}$, we map it to edge $\overrightarrow{H(s), H(d)}$ in the graph sketch, and use square hashing method to find the r^2 mapped buckets and check them one by one. Once we find a bucket in row $h_{i_s}(s)$ and column $h_{i_d}(d)$ which contains the fingerprint pair $\langle f(s), f(d)\rangle$ and the index pair $\langle i_s, i_d\rangle$, we return its weight as the result. If we find no result in the r^2 buckets, we search the buffer for edge $\overrightarrow{H(s), H(d)}$ and return its weight. If we still cannot find it, we return -1.

1-hop Successor Query: To find the 1-hop successors of a node v, we map it to node $H(v)$ in G_h. Then we compute its hash address sequence according to $H(v)$, and check the r rows with index $\{h_i(v)|1 \leqslant i \leqslant r\}$. If a bucket in row $h_{i_s}(v)$, column C contains fingerprint pair $\langle f(v), f(u)\rangle$ and index pair $\langle i_s, i_d\rangle$, where $f(u)$ is any integer in range $[0, F)$ and i_d is any integer in range $[1, r]$, we use $f(u)$, i_d and C to compute $H(u)$ as stated above. Then we add $H(u)$ to the 1-hop successor set SS. After searching the r rows, we also need to check the buffer to see if there are any edges with source node $H(v)$ and add their destination nodes to SS. We return -1 if we find no result. Otherwise we obtain the original node ID from SS by accessing the hash table which stores $\langle H(v), v\rangle$.

1-hop Precursor Query: To answer an 1-hop precursor query, we have the analogue operations with 1-hop successor query if we switch the columns and the rows in the matrix X. The details are omitted due to space limit.

After applying square hashing, the edges with source node $H(v)$ in G_h are on longer stored in a single row, but spread over r rows with addresses $\{h_i(v)|1 \leqslant i \leqslant r\}$. Similarly, edges with destination node $H(v)$ are stored in r different columns. These rows or columns are shared by edges with different source nodes or destination nodes. The higher degree a node has, the more buckets its edges may take. This eases congestion brought by the skewed node degree distribution. Moreover, as each edge has multiple mapped buckets, it has a higher probability to find an empty one. Obviously, square hashing will reduce the number of *left-over edges*.

5.2 More Optimizations

5.2.1 Dealing with edge deletion

In this section, we extend GSS to graph streams with edge deletions. We suppose the weight w in each item $(\overrightarrow{s,d};t;w)$ in the graph stream can be negative, which means to delete a former item. When the weight of an edge in the streaming graph is decreased to 0. It is deleted. [2] In this case the update operator becomes as follows:

Update with edge deletion: When a new item $(\overrightarrow{s,d};t;w)$ comes in the graph stream S, we map it to edge $\overrightarrow{H(s),H(d)}$ in the graph sketch G_h. Then we find the r^2 mapped buckets and check them one by one. If we find a mapped bucket with fingerprint pair and index pair equal to the corresponding pairs of $\overrightarrow{H(s),H(d)}$, we add w to the weight in it. If the weight becomes 0, we clear this bucket. If we find no mapped bucket with matched fingerprint pair and index pair, we further check the buffer for $\overrightarrow{H(s),H(d)}$. If we find it in the buffer, we add w to its weight, and remove the edge if its weight becomes 0. If we can neither find this edge in the mapped bucket or the buffer, we insert it into the first empty mapped bucket. If there is no empty mapped bucket, we add it into the buffer.

Notice that different from update operator in Section 5.1. When we meet an empty mapped bucket, we cannot directly insert $\overrightarrow{H(s),H(d)}$ into the bucket and end the update procedure. Because $\overrightarrow{H(s),H(d)}$ may have arrived before, but at that time, this bucket is occupied by another edge, which is deleted later. Thus $\overrightarrow{H(s),H(d)}$ may be stored in latter mapped buckets or the buffer. Therefore, we have to check all the mapped buckets and the buffer to find out if $\overrightarrow{H(s),H(d)}$ has arrived or not. As a result, when dealing with deletion, the update speed of GSS will become lower. But with the mapped bucket sampling technique described in the following section, the decrement is not large. We evaluate the update speed of GSS both with and without deletion in Section 8.6.

5.2.2 Mapped Buckets Sampling

In the modified version of GSS, each edge has r^2 mapped buckets. In the worst case of an updating, we may need to check all r^2 mapped buckets. We can use a sampling technique to decrease the time cost. Instead of check all the r^2 buckets, we select k buckets as a sample from the mapped buckets. We call these buckets *candidate buckets* for short. For each edge we only check these k buckets in updating and queries, and the operations are the same as above. The method to select these k buckets for an edge e is also a linear congruence method, with the sum of the source node fingerprint and the destination node fingerprint as seed.

5.2.3 Multiple Rooms

When the memory is sufficient, we can separate each bucket in the matrix into l segments, and each segment contains an edge. We call each segment a *room* for convenience. When performing the basic operators, we use the same process as above to find the buckets we need to check, and search all the rooms in them to find qualified edges or empty rooms. Compared to enlarging the matrix and select more candidate buckets, using multiple rooms in each bucket has a higher locality, as these adjacent rooms can be fetched in fewer memory accesses. This is fully utilized when implementing GSS in hardwares like FPGA, as will be discussed in details in Section 7.3.

2. directly removing an edge $e = \overrightarrow{s,d}$ is equal to receiving an item $(\overrightarrow{s,d};t;-w(e))$ where $w(e)$ in the weight of e in current streaming graph

6 Analysis

6.1 Accuracy Analysis

Recall that GSS uses two steps to summarize a graph stream. In the first step, it uses a hash function $H(\cdot)$ to compress the original streaming graph G into a graph sketch G_h. In the second step, it stores the graph sketch with a novel data structure. In the following sections, we will first prove that the second step has no error. Then we will analyze the error brought by the first step

First we prove that the storage of the graph sketch G_h in the data structure of GSS is accurate. As the buffer is an adjacency list that stores edges in G_h accurately, we only need to check the matrix. We need to prove that each occupied bucket of the matrix is uniquely possessed by one edge $\overrightarrow{H(s),H(d)}$ in G_h. In other words, we need to prove the following theorem:

Theorem 1. With the position and the content of a bucket in the matrix of GSS, we can get a unique solution about the ID of the stored edge.

As discussed in Section 5.1, given the position and content of a bucket, we can recover a unique edge ID $\overrightarrow{H(s),H(d)}$. The recovery procedure is discussed in detail in Section 5.1, we omit it here to save space. When there are multiple rooms in a bucket, though they share the same position, they have different fingerprint pairs and index pairs. Therefore, the storage of the graph sketch G_h is accurate.

Second, we analyze the error in the procedure of mapping G to G_h. We use $\bar{P}(e_1,e_2)$ to represent the probability of the following event:

Definition 5. **Edge Collision**: An edge collision between streaming graph edge e_1 and e_2 means $H(e_1)=H(e_2)$ in the graph sketch G_h.

If e_1 and e_2 share no endpoints, we have $\bar{P}(e_1,e_2)=\frac{1}{M^2}$. Otherwise, if they have the same source / destination node, we have $\bar{P}(e_1,e_2)=\frac{1}{M}$, because they will collide if their destination / source nodes have the same hash value.

Next, we will analyze the accuracy of the query primitives with this probability.

For edge query, if the queried edge collides with other edges, the query result will be larger than the true value. We use $Adj(e)$ to denote set of edges that share one endpoint with e, and the sum of their weight is $W_{Adj(e)}$. E denotes edges in the streaming graph G, but excluds e if e is in the streaming graph. Note that the queried edge may not exist in the streaming graph, and in this case the result of edge query is -1. Then $E-Adj(e)$ denotes the edges that share no endpoints with e, we use $W_{E-Adj(e)}$ to represent sum of their weight. The query result is correct if and only if all

other edges does not collide with e. For edges in $Adj(e)$, the probability that all of them do not collide with e is

$$Pr_1 = (1 - \frac{1}{M})^{|Adj(e)|} = e^{-\frac{|Adj(e)|}{M}} \tag{4}$$

For edges in $E - Adj(e)$, the probability that all of them do not collide with e is

$$Pr_2 = (1 - \frac{1}{M^2})^{|E-Adj(e)|} = e^{-\frac{|E-Adj(e)|}{M^2}} \tag{5}$$

The correct rate of edge query is

$$CorrectRate(e) = Pr_1 \times Pr_2 = e^{-\frac{|E|+(M-1)\times|Adj(e)|}{M^2}} \tag{6}$$

The expected error of edge query, namely the difference between the queried result and the true value, is the sum of weight of all edges that collide with e. which is

$$Error(e) = \frac{W_{Adj(e)}}{M} + \frac{W_{E-Adj(e)}}{M^2} \tag{7}$$

For 1-hop successor query, we use $Suc(v)$ to represent the successors of the query node v, and use $Pre(u)$ to represents the precursor set of a node u. Because the true successors will definitely be reported, the potential error is including false successors. The successor query result of v is correct only if for each node u in $V - Suc(v)$, edge $\overrightarrow{v,u}$ is correctly reported as not existent. This probability is:

$$\begin{aligned} CorrectRate_{suc}(v) &= \prod_{u\in V-Suc(v)} CorrectRate(\overrightarrow{v,u}) \\ &= \prod_{u\in V-Suc(v)} e^{-\frac{|E|+(M-1)\times|Adj(\overrightarrow{u,v})|}{M^2}} \\ &\approx e^{-\frac{[|E|+(M-1)\times(|Suc(v)|+\frac{|E|}{|V|})]\times(|V|-|Suc(v)|)}{M^2}} \end{aligned} \tag{8}$$

The precision of the 1-hop successor query, namely the ratio of the true successors against the reported successors, is

$$Precision_{suc}(v) = \frac{|Suc(v)|}{|Suc(v)| + FalseSuc(v)} \tag{9}$$

where $FalseSuc(v)$ represent the number of false successors, and the expected value is :

$$\begin{aligned} FalseSuc(v) &= \sum_{u\in V-Suc(v)} (1 - CorrectRate(\overrightarrow{v,u})) \\ &= \sum_{u\in V-Suc(v)} (1 - e^{-\frac{|E|+(M-1)\times(|Suc(v)|+|Pre(u)|)}{M^2}}) \end{aligned} \tag{10}$$

The 1-hop precursor query is similar to 1-hop successor query. We only need to exchange the $Suc(\cdot)$ and $Pre(\cdot)$ function in the formula. From the formulas we can see that the larger M is, the higher the accuracy is. In GSS we have $M = m \times F$, where m is the width of the matrix, and F is the maximum size of the fingerprints. For a matrix with $m = 1000$ and 16-bit fingerprint, M can be as larger as 65536000, This guarantees the accuracy. On the other hand, in TCM the accuracy analysis is the same as GSS, but we have $M = m$. This lead to the difference in accuracy.

6.2 Buffer Size Analysis

After all the improvements, the buffer in GSS is very small. The mathematical expression of the buffer size is very complicated and is influenced by many details of the graph. Therefore we give an expression of the probability that a new edge e becomes a *left-over edge*, which means inserted into the buffer, as a measurement. We use E to denote edges already in the streaming graph before e arrives. The number of edges in E is denoted as $|E| = N$ for simplicity of presentation in the following analysis. $Adj(e)$ denotes edges that have common source node or common destination node with e. We suppose $|Adj(e)| = D$ for simplicity. The width of the matrix is m, and each bucket in the matrix has l rooms. For each node we compute a hash address sequence with length r. For each edge we choose k candidate buckets among the r^2 mapped buckets.

For each candidate bucket of e, as the edges in $E-Adj(e)$ are randomly inserted into the matrix with area m^2, the probability that there are a_1 non-adjacent edges inserted into it is:

$$\begin{aligned} p_1(a_1) &= \binom{N-D}{a_1} \times (\frac{1}{m^2})^{a_1} \times (1 - \frac{1}{m^2})^{N-D-a_1} \\ &= \binom{N-D}{a_1} \times (\frac{1}{m^2})^{a_1} \times e^{-\frac{N-D-a_1}{m^2}} \end{aligned} \tag{11}$$

As the D adjacent edges in $Adj(e)$ are randomly inserted in an area of $r \times m$ (r length-m rows mapped by the source node of e or r length-m columns mapped by the destination node of e), the probability that there are a_2 adjacent edges inserted into this bucket is:

$$\begin{aligned} p_2(a_2) &= \binom{D}{a_2} \times (\frac{1}{r\times m})^{a_2} \times (1 - \frac{1}{r\times m})^{D-a_2} \\ &= \binom{D}{a_2} \times (\frac{1}{r\times m})^{a_2} \times e^{-\frac{D-a_2}{r\times m}} \end{aligned} \tag{12}$$

The probability that there are already n edges inserted into this bucket is:

$$p(n) = \sum_{a=0}^{n} p_1(a) \times p_2(n-a) \tag{13}$$

The probability that there are less than l edges inserted into this bucket is:

$$\begin{aligned} Pr &= \sum_{n=0}^{l-1} p(n) \\ &= \sum_{n=0}^{l-1}\sum_{a=0}^{n} p_1(a) \times p_2(n-a) \\ &= \sum_{n=0}^{l-1}\sum_{a=0}^{n} \binom{N-D}{a}\binom{D}{n-a} (\frac{1}{m^2})^{a} (\frac{1}{rm})^{n-a} e^{-(\frac{N-D-a}{m^2}+\frac{D-n+a}{rm})} \end{aligned} \tag{14}$$

This is also the lower bound that the bucket is still available for e. The probability that e can not be inserted into the matrix is the probability that all the k candidate buckets are not available, which is:

$$P = (1 - Pr)^k \tag{15}$$

Notice that this is an upper bound as we ignore collisions in the map procedure from G to G_h.

This left-over probability is rather small. In Appendix B of the supplementary materials, we demonstrate the curve of left-over probability. According to the figure, the left-over probability i below 1% in most times. Experiment in Section 8.5 also confirms the small buffer size.

6.3 Time and Memory Cost Analysis

In this section, we analyze the space cost of GSS and the time cost of primitives. The memory cost of GSS is $O(|E_h|+|B|)$, where $|E_h|$ is the number of edges in the graph sketch G_h and $|B|$ is the size of buffer, which are both below $O(E)$. When we use a hash table to store the original node IDs, additional $O(|V|)$ memory is needed, but the overall memory cost is still $O(|E|)$. The update time cost is $O(k+\frac{|B|}{|E_h|}|B|)$, where k is the number of sampled buckets. When an edge is stored in the matrix, we only need to check at most k candidate buckets, which takes $O(k)$ time. Each edge has probability $\frac{|B|}{|E_h|}$ to be stored in the buffer. When it is stored in the buffer, the update takes additional $O(|B|)$ time, as the buffer is an adjacency list. From the analysis in Section 6.2, we know that the buffer size $|B|$ is very small. Therefore the time cost of this part is also small.

The time cost of queries is based on the algorithms we use. We consider the time cost of the primitives as an evaluation. The time cost of the edge query primitive is the same as the update, and the time cost of the 1-hop successor query and 1-hop precursor query is $O(rm+|B|)$ in GSS, where m is the width of the matrix and r is the length of the hash address sequence. Because we have to scan r rows / columns and the buffer to find all successors / precursors of a node. In Section 7, we will propose more improvements and decrease the time cost of 1-hop successor query and 1-hop precursor query to $O(\frac{m}{r}+|B|)$.

7 Accelerating Query Primitives

Although GSS has achieved high update speed and small memory usage, it still has performance issues. During successor query and precursor query, we need to scan the mapped rows or columns of a node, which is time consuming. In order to decrease the cost, we propose an improvement on the layout of the matrix. We partition the matrix of GSS into multiple blocks, and the improved version is called *blocked GSS*. Details will be discussed in Section 7.1. Based on the blocked version, we propose two directions of accelerating: GSS with node bitmaps in Section 7.2 and GSS implemented with FPGA (Field Programmable Gate Array) in Section 7.3.

7.1 Blocked GSS

In the blocked version of GSS, we divide the matrix into $r\times r$ blocks. These blocks are organized as r rows and r columns, as shown in Figure 5. In order to distinguish the rows / columns of blocks with the rows / columns of buckets in each block, we represent the block row with BR and block column with BC.

For each node $H(v)$ in the graph sketch, we generate an address list $\{h_i(v)|1\leqslant i\leqslant r\}$ for it with the same procedure

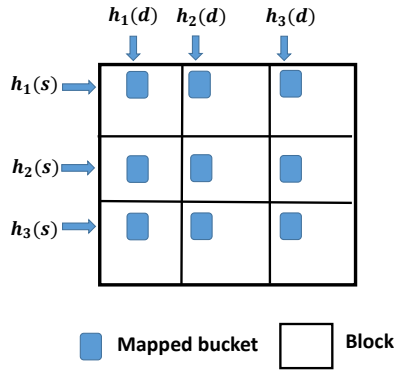


Fig. 5. The Matrix of blocked GSS

as Equation 1 and 2. The only difference is that the value range of the addresses is $[0,\frac{m}{r})$ rather than $[0,m)$. For each edge $\overrightarrow{H(s),H(d)}$ in the graph sketch, we map it to $r\times r$ mapped buckets, one in each block. In the block of BR i_s and BC i_d ($1\leqslant i_s\leqslant r$, $1\leqslant i_d\leqslant r$), the mapped bucket is located in row $h_{i_s}(s)$ and column $h_{i_d}(d)$. Then we select candidate buckets from the mapped buckets and store the edge in the first empty candidate bucket. If all candidate buckets are occupied, it is stored in the buffer.

Compared to non-partition version, there are also some other differences in blocked GSS. When storing an edge, we only store the fingerprint pair and the edge weight. The index pair is not needed. Because the location of the block implies the indexes $\langle i_s,i_d\rangle$. However, in order to keep the value range M of the map function $H(\cdot)$ not changed, the fingerprint has to be $log(r)$ bits longer, as the range of the hash address is r time smaller. Therefore the memory usage does not change compared to the non-partition version.

7.2 Node bitmap

As stated above, most nodes in the graph have low degrees. Their neighbors are stored in only a few blocks, and we can record these blocks to narrow the search area in precursor or successor query. For the 1-hop successor query, we can use a bitmap to record whether a block stores successors of a node. For each node $H(v)$ in the graph sketch, we assign a bitmap with $r\times r$ bits. The i_{th} bit is set to 1 if the block in BR i/r and BC $i\%r$ stores the successors of $H(v)$. Otherwise it is set to 0. It is the similar in the 1-hop precursor query. The bitmaps can be stored in the same hash table which stores the node IDs. Each node has $2\times r^2$ bits more memory usage as a cost. We call GSS with such node bitmaps GSS_{nb}. With these bitmaps , we can only check the blocks whose corresponding bits are 1 in the 1-hop successor or precursor query, which omits lots of unnecessary scan.

We also propose an alternative solution which uses less memory, but also achieves less speed improvement. In this solution, we use two r-bit bitmaps for each node in successor query, corresponding to the block rows and block columns, respectively. If an out edge of node $H(v)$ in the graph sketch is inserted in BR i and BC j, we set the i_{th} bit of the first bitmap and the j_{th} bit of the second bitmap to 1. In 1-hop successor query, we check a block in BR i amd BC j if and only if both the i_{th} bit of the first bitmap and the j_{th} bit of the second bitmap are 1. It is similar for the

1-hop precursor query. We call GSS with such short bitmaps GSS_{sb} for short. In GSS_{sb}, we cannot know exactly which block contains neighbors of a node, thus may perform some vain scans. The speed of the 1-hop precursor and successor query primitive will be 2-3 times slower than GSS_{nb}, as will be shown in Section 8.7.

Though GSS_{nb} and GSS_{sb} has higher query speed compare to original GSS, they do not support edge deletions. We cannot find out when to reset a bit in the node bitmap to 0, as we cannot determine when a block, or row / column does not contain neighbors of a node any more. In the next section, we will propose another acceleration solution, which supports deletion without update speed loss.

7.3 Acceleration with FPGA

FPGA (Field Programmable Gate Array) plays an important role in hardware acceleration because of its high parallelism and low energy consumption. The matrix structure of GSS leads to a high fitness with FPGA. We can use FPGA to accelerate the query primitives of GSS. An FPGA acceleration board is composed of a chip and multiple memory banks. Each memory bank has multiple ports connected with the chip, allowing memory accesses in parallel. Nowadays, the global memory provided by the memory banks can be as large as 64 GB, which makes it possible to store the sketch of large graphs.

When implementing GSS on FPGA (we call it GSS-FPGA for short), we map the edges in the graph stream into edges in the graph sketch with an encoder on the CPU host, and place the matrix and the buffer in the global memory of FPGA. Update and query are performed with kernels on the FPGA chip. In order to achieve high parallelism, we separate the blocks of the matrix among multiple memory banks, and each block is bounded with an independent port, so that we can access these blocks in parallel. As the number of ports is limited on FPGA, we can not divide the matrix into a large number of blocks. In order to keep a high loading rate of the matrix, We can enlarge each bucket into multiple rooms as a compensation. Using multiple rooms also helps to make full use of the port width. Because the port width is 512-bit, we can fetch multiple adjacent rooms in one memory access.

In GSS-FPGA, all the primitives can benefit from the parallelism. In the 1-hop successor / precursor query, we can scan different blocks in parallel, and in each block, buckets in the mapped row / column can be checked in pipeline. Therefore, we can achieve high speed without the cost of storing bitmaps. The time cost of the 1-hop successor / precursor query is reduced to $O(\frac{m}{r} + |B|)$. In update and edge query, the mapped buckets of an edge can also be checked in parallel. Especially, in the edge query primitive, multiple queries can be processed in pipeline, which leads to much higher throughput compared with CPU implementation. On the other hand, in the update primitive, the read of mapped buckets and write back of them after updates induce read-write lock, which prevents the update primitive from being fully pipelined. Due to the low frequency of FPGA, the update primitive is slower than CPU implementation. But we can still achieve a speed of 1.7 million updates per second, as shown in Section 8.7. Notice that because we can scan all the mapped buckets in parallel, we do not need to select candidate buckets in GSS-FPGA.

8 Experimental Evaluation

In this section, we show our experimental studies of GSS. In Section 8.4, we evaluate the accuracy of GSS in three graph query primitives and compare it with TCM. In Section 8.5, we evaluate the buffersize of GSS. In Section 8.6, we compare the update speed and memory usage of GSS with TCM, adjacency lists and an incremental lossless graph summarization method MoSSo [3]. In Section 8.7, we further evaluate the optimizations proposed in Section 7. At last, we evaluate the performance of GSS in graph analytic mission Single Source Shortest Path (SSSP) in Section 8.8. In the supplementary materials D, we present experiments on other compound queries like triangle counting, reachability query and subgraph matching. All experiments are performed on a server with dual 18-core CPUs (Intel Xeon CPU E5-2697 @2.3 GHz, 2 threads per core) and 192 GB DRAM memory, running CentOS. All algorithms including GSS and TCM are implemented in C++. The codes are open sourced [42]. The code for MoSSo is provided by the original authors at http://dmlab.kaist.ac.kr/mosso/.

8.1 Datasets

1)**lkml-reply**[3].The first dataset is a collection of communication records in the network of the Linux kernel mailing list. It contains 63399 email addresses (nodes) and 1096440 communication records (edges). 2)**networkflow**. The second dataset is a collection of network packets downloaded from a backbone router. It contains 445440480 communication records (edges) concerning 2601005 different IP addresses (nodes). 3)**Twitter**[4].The third dataset is a network contains Twitter follow data based on a snapshot taken in 2009. Each node represents a user and each directed edge indicates that a user follows another user. The original dataset is a static dataset with no duplication, and it contains $52,579,682$ nodes and $1,963,263,821$ edges. The highest node degree in this dataset reaches $3,691,240$. We randomly generate duplication for its edges with zipf distribution. The duplicated dataset contains $3,720,775,389$ edges. For all the three datasets, edges are weighted by their frequencies in the dataset. We feed edges to the data structure in random orders to simulate graph streams.

8.2 Metrics

Average Relative Error (ARE): ARE measures the accuracy of reported weight in edge queries and reported distance of node pairs in SSSP. Given a query q, the *relative error* is defined as: $RE(q) = \left|\frac{\hat{f(q)}}{f(q)} - 1\right|$. $f(q)$ and $\hat{f(q)}$ are the real answer and the estimated value of q. When giving a query set, the *average relative error* (ARE) is measured by averaging the relative error over all queries in it. A more accurate data structure has smaller ARE.

Average Precision: We use average precision as the evaluation metric in 1-hop successor / precursor queries. Given such a query q, we use SS to represent the accurate set of 1-hop successors / precursors of the queried node, and $\hat{SS}$ to represent the set we get by q. As TCM and GSS have only false positives, which means $SS \subseteq \hat{SS}$, we

3. http://konect.uni-koblenz.de/networks/lkml-reply
4. http://konect.cc/networks/twitter_mpi/

define the precision of q as $Precision(q) = \frac{|SS|}{|\hat{SS}|}$. Average precision of a query set is the average value of the precision of all queries in it. A more accurate data structure has higher *Average Precision*

Compression Ratio: It measures the effectiveness of graph compression, defined as 1 - (memory usage after compression)/(original memory usage with adjacency lists).

Buffer Percentage: It measures buffer size of GSS. *Buffer percentage* is defined as the number of edges in the buffer divided by the total number of edges in the graph stream.

8.3 Experiments settings

In experiments, we implement two kinds of GSS with different fingerprint sizes: 12 bits and 16 bits, and vary the matrix size. We use f to represent the fingerprint size. We apply all improvements to GSS, and the parameters are as follows. Each bucket in the matrix contains $l = 8$ rooms. For lkml-reply and networkflow, the length of the address sequences is $r = 8$, and the number of *candidate buckets* for each edge is $k = 4$. For Twitter which is highly skewed, we set $r = 16$ and $k = 8$. As for TCM, we apply 4 graph sketches to improve its accuracy. Its memory usage is 8 times larger than GSS in lkml-reply and networkflow, and the same as GSS in Twitter (due to short of memory). The hash tables used in TCM and GSS to store $\langle H(v), v\rangle$, namely hash value - original ID pairs are classic hash tables with linked lists to address hash collisions. IDs with the same hash value is organized as a linked list in the same key-value pair.

8.4 Experiments on Query Primitives

In this section, we evaluate the accuracy of GSS in the 3 basic graph query primitives. Figure 6, Figure 7, and Figure 8 show that ARE of edge queries and average precision of 1-hop precursor / successor queries respectively. We only show the result of Twitter in 1-hop precursor / successor queries due to space limitation. The edge query set contains all edges in the graph stream, and the 1-hop precursor / successor query set contains all nodes in the graph stream. The results tell us that GSS performs much better in supporting these query primitives than TCM, especially in the 1-hop precursor / successor query primitives. In fact the precision of 1-hop precursor / successor queries of TCM in a dataset as large as Twitter is nearly 0. On the other hand, with 16-bit fingerprint GSS can always get ARE below 10^{-2} in edge queries and average precision beyond 90% in 1-hop precursor / successor queries.

8.5 Experiments on Buffer Size

In this section, we evaluate the buffer size of GSS. Figure 9 shows the buffer percentage in Twitter. The five curves in the figure represent 1) GSS without square hashing (no-SH). 2) GSS with square hashing and $k = 2/4/8$ candidate buckets. The x-label is the width of the matrix. From the curves, we can see that square hashing significantly decreases the buffer size. With the increment of candidate bucket number k, the buffer size also decreases, but the gap shrinks as the matrix size grows. Besides, with $k \geqslant 4$ and matrix width lager than 18000, the buffer size is smaller than 2%. It means in most updates we do not need to use the buffer.

TABLE 1
Update Speed (Mops)

Data Structure	lkml-reply	networkflow	Twitter
GSS	5.1	5.67	1.58
GSS(with deletion)	5	3.93	1.04
TCM	1.4	0.03	TLE
TCM(without hash table)	6.2	6.1	2.7
Adjacency Lists (Successor)	0.43	1.7	0.03
Adjacency Lists (Precursor)	0.33	0.04	TLE
MoSSo	0.012	0.005	TLE

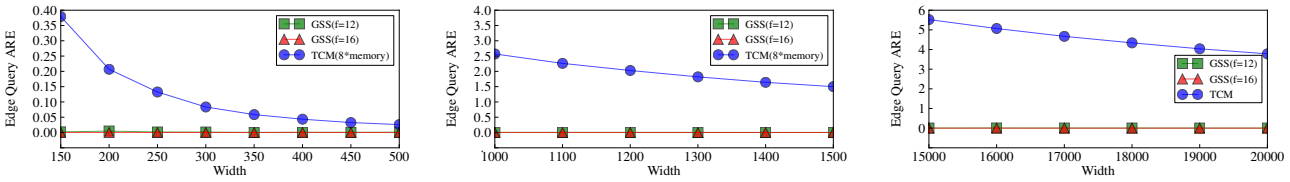
8.6 Memory and Speed Evaluation

In this section we evaluate the memory usage and update speed of GSS. We compare the update speed of GSS, TCM, adjacency lists and the state-of-the-art incremental lossless graph summarization method MoSSo in Table 1. We compare the memory usage of GSS and adjacency lists, and the result is shown in Table 2. We also compare the compression ration of MoSSo and GSS in Table 3.

For GSS we set the matrix width to be 200 in lkml-reply, 1200 in networkflow, and 18000 in Twitter. The fingerprint length is 16, and other parameters are the same as above. Former experiments show that GSS achieves nearly accurate query results with these parameters. TCM uses 8 times memory compared to GSS in lkml-reply and networkflow, and the same memory as GSS in Twitter. Both GSS and TCM use hash tables to store the original node IDs, but we also present the speed of TCM without hash table. For MoSSo, the parameters are set according to the recommendation of the authors (escape probability set to 0.3 and number of samples set to 120), details can be referred in [3]. As MoSSo does not support duplicate edges and directed graphs, we remove the edge duplication and edge direction in the three datasets for MoSSo. For adjacency lists, in order to support both the successor query and the precursor query, 2 sets of lists are needed for each graph. The first set stores the successor lists, and the second set stores the precursor lists. In Table 1, we show both the update speeds of the successor lists and the precursor lists. In Table 2. The memory usage is the sum of both sets of lists.

Table 1 shows the update speed of the algorithms. The algorithms are accelerated with -O2 option of GCC. In each dataset, we insert all the edges into the data structure and calculate the average speed. The unit we use is Million Operations per Second (Mops). If an algorithm cannot finish processing the dataset in 48 hours, the result is marked as Time Limit Exceeded (TLE).

From the table, we can see that GSS always achieve an update speed over $1Mops$. The update speed is lower in Twitter. Because this dataset is so large that even if only $1\% \sim 2\%$ edges are stored in the buffer, updating them still has a high cost and brings decrement in speed. We also evaluate the speed of GSS with edge deletions. We insert all the edges in each dataset into GSS twice, with positive weight at the first time and negative weight at the second time. GSS processes these deletion-included updates with update method discussed in Section 5.2.1, and we



(a) lkml-reply (b) networkflow (c) Twitter

Fig. 6. Average Relative Error of Edge Queries

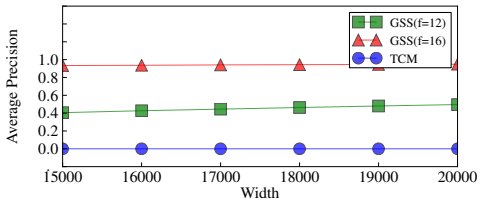


Fig. 7. Average Precision of 1-hop Precursor Queries

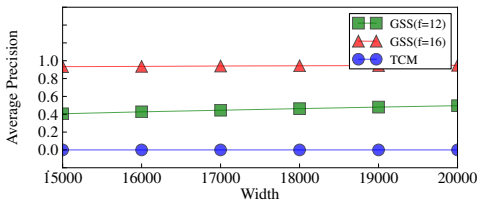


Fig. 8. Average Precision of 1-hop Successor Queries

compute the average speed. As discussed above, the speed will have a decrement with deletion, as we have to scan all the candidate buckets and the buffer in each update. But its speed is still higher than other algorithms.

TCM has a sharp decrement in speed with the graph size growing. Because for large graphs, the map range of TCM is limited while the node set is large, resulting into a lot of node IDs mapped to the same hash value. In each update of the hash table, we have to scan a long ID list attached to the same hash value. It leads to low update speed. As a comparison, we also present the update speed of TCM without storing node IDs with hash tables. We can find that in the case its update speed is comparable to GSS. But without the hash table it cannot support queries requiring node IDs like the 1-hop successor query.

The adjacency list is sensitive to the skewness. Its update speed varies through different datasets. Even for the same dataset, the update speed of the successor list and the precursor list may also have a large difference. And in large graphs like Twitter, the update speed is below $0.03Mops$.

For MoSSo, as it has to analyse the graph topology to find nodes with similar neighborhood, its update speed is much lower than the adjacency lists and below $0.01Mops$ in most times, which cannot meet the demand of high throughput of graph streams.

Table 2 shows the memory usage of GSS and adjacency lists. We do not show the memory usage of TCM, as in

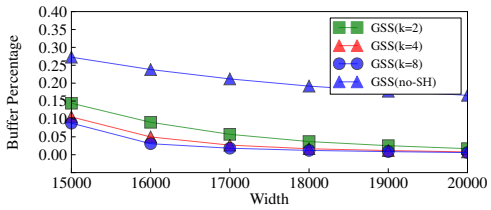


Fig. 9. Buffer Percentage of GSS

TABLE 2
Memory Usage(MB)

Data Structure	lkml-reply	networkflow	Twitter
GSS	3.36	161	2.6×10^4
GSS_{sb}	3.41	166	2.62×10^4
GSS_{nb}	3.79	202	2.97×10^4
Adjacency Lists	8.76	353	6.47×10^4

experiments it is set according to the memory usage of GSS. From the table we can see that the memory usage of GSS is $30\% \sim 50\%$ of the adjacency lists. This memory usage includes the memory used by the hash table which stores the original node IDs. The memory usage of GSS_{nb} and GSS_{sb} is also shown in the table, we can see that GSS_{nb} needs 25% additional memory at most compared to the original GSS, while GSS_{sb} barely needs any additional memory.

Because MoSSo can only support undirected graphs, we have to remove edge directions for it in all three datasets. After that the streaming graphs become much smaller (as edge $\overrightarrow{s,d}$ and $\overrightarrow{d,s}$ will be combined). Therefore we do not directly compare the memory usage of MoSSo with GSS. We compare their compression ratio in Table 3 instead. From the table we can see that though GSS has small errors, but its compression ratio is higher than MoSSo, and its update speed is also much higher (Table 1).

8.7 Experiment of Optimizations

In this section, we evaluate the effect of optimizations proposed in Section 7. We evaluate the speed of four primitives, update, edge query, 1-hop successor query and 1-hop precursor query of GSS with different optimizations,

TABLE 3
Compression Ratio

Data Structure	lkml-reply	networkflow	Twitter
GSS	61.6%	54.4%	59.7%
MoSSo	17.4%	45.5%	TLE

TABLE 4
Speed of Different Versions of GSS (Mops)

Data Structure	Update	Edge Query	Successor Query	Precursor Query
GSS	5.1	4.9	0.06	0.04
GSS_{nb}	4.3	5.5	0.36	0.32
GSS_{sb}	4.8	5.7	0.19	0.17
GSS-FPGA	1.7	9.6	0.31	0.24

TABLE 5
ARE of GSS in SSSP

lkml-reply	networkflow	Twitter
4.28×10^{-4}	1.97×10^{-3}	5.83×10^{-3}

and the result is shown in Table 4. The unit of speed is Million Operations per Second (Mops). The dataset we use in experiments is lkml-reply. Parameters for 3 versions of GSS on CPU are the same as Section 8.6. On the other hand, GSS-FPGA is implemented on FPGA board Xilinx u280. The board has 32 HBM memory banks, with totally $8GB$ memory. The matrix of GSS is split into 9 blocks, separated on 9 independent memory banks for parallelism. Each bucket of the matrix has 8 rooms. The total memory usage of the matrix is the same as the CPU versions. From Table 4. We can see that the original GSS has the lowest speed in successor and precursor queries. For GSS_{nb} or GSS_{sb}, the update speed and the edge query speed is similar with the original version, but the successor query and the precursor query are $6 \sim 8$ and $3.2 \sim 4.25$ times faster than the original version. When implemented on FPGA, the update speed of GSS decreases due to the low frequency of FPGA. However, as the edge query is fully pipelined, the speed is more than 1.68 times higher than the CPU implementations. Both the successor query and the precursor query have a speed competitive with the speed of the optimized GSS on CPU. Besides, FPGA-GSS can support deletions without decrement in update speed, while GSS_{nb} and GSS_{sb} do not support edge deletions.

8.8 Experiment of SSSP

In this section we evaluate the performance of GSS in graph analytic mission Single Source Shorted Path (SSSP). In lkml-reply and networkflow, we use nodes with top-100 degrees as source nodes and carry out SSSP computation 100 times. In Twitter, we use nodes with top-10 degrees as source nodes and carry out SSSP computation 10 times, as the graph is large and SSSP computing in it is time consuming. The SSSP computation is carried out with Dijkstra algorithm. We compute the Average Relative Error (ARE) of the estimated distance between each node in the graph and the source node. We present the result in Table 5. The result shows that GSS always has an ARE below 1%. We also show the average execution time of SSSP with adjacency lists and GSS in Table 6 (GSS_{nb} is used). Due to the matrix structure, GSS has a lower speed in topology queries. It has to scan rows or columns in the matrix to get successors / precursors. But GSS can still finish SSSP, which has a time complexity of $O(|E|log(|V|))$, in graph as large as billions of edges (Twitter) in about 4 hours. We believe the drawback in query speed is a necessary cost of high update speed and small memory usage. GSS is suitable for situations where the demand on update speed and memory consumption is the major concern, like network measurement in routers.

TABLE 6
Execution Time of SSSP (Seconds)

Data Structure	lkml-reply	networkflow	Twitter
GSS	0.1	7.68×10^{-3}	1.36×10^{4}
Adjacency Lists	0.057	1.56×10^{-3}	975

9 Conclusion

Graph stream summarization is a problem rising in many fields. However, as far as we know, there is no prior work with high update speed, small memory usage and high accuracy. In this paper, we propose graph stream summarization data structure Graph Stream Sketch (GSS). It has $O(|E|)$ memory usage and high update speed. It supports most queries based on graphs and has accuracy which is higher than state-of-the-art by magnitudes. Both mathematical analysis and experiment results confirm the superiority of our work.

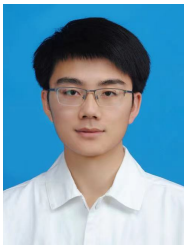
References

[1] S. Guha and A. McGregor, "Graph synopses, sketches, and streams: A survey," *PVLDB*, vol. 5, no. 12, pp. 2030–2031, 2012.
[2] W. Fan, J. Li, X. Wang, and Y. Wu, "Query preserving graph compression," in *SIGMOD*, pp. 157–168, ACM, 2012.
[3] J. Ko, Y. Kook, and K. Shin, "Incremental lossless graph summarization," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 317–327, 2020.
[4] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," in *Latin American Symposium on Theoretical Informatics*, pp. 29–38, 2004.
[5] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *SIGMOD*, pp. 1449–1463, ACM, 2016.
[6] D. Thomas, R. Bordawekar, C. C. Aggarwal, and S. Y. Philip, "On efficient query processing of stream counts on the cell processor," in *ICDE*, pp. 748–759, IEEE, 2009.
[7] N. Tang, Q. Chen, and P. Mitra, "Graph stream summarization: From big bang to big crunch," in *SIGMOD*, pp. 1481–1496, 2016.
[8] A. Khan and C. Aggarwal, "Query-friendly compression of graph streams," in *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pp. 130–137, 2016.
[9] S. Raghavan and H. Garcia-Molina, "Representing web graphs," in *ICDE*, pp. 405–416, IEEE, 2003.
[10] M. Riondato, D. García-Soriano, and F. Bonchi, "Graph summarization with quality guarantees," *Data mining and knowledge discovery*, vol. 31, no. 2, pp. 314–349, 2017.
[11] D. Peleg and A. A. Schäffer, "Graph spanners," *Journal of graph theory*, vol. 13, no. 1, pp. 99–116, 1989.
[12] D. A. Spielman and N. Srivastava, "Graph sparsification by effective resistances," *SIAM Journal on Computing*, vol. 40, no. 6, 2011.
[13] S. Assadi, S. Khanna, and Y. Li, "On estimating maximum matching size in graph streams," in *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1723–1742, SIAM, 2017.
[14] M. Kapralov, A. Mousavifar, C. Musco, C. Musco, N. Nouri, A. Sidford, and J. Tardos, "Fast and space efficient spectral sparsification in dynamic streams," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1814–1833, SIAM, 2020.
[15] K. J. Ahn, S. Guha, and A. McGregor, "Graph sketches: sparsification, spanners, and subgraphs," in *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pp. 5–14, 2012.

[16] S. Bhattacharya, M. Henzinger, D. Nanongkai, and C. Tsourakakis, "Space-and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams," in *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pp. 173–182, 2015.
[17] O. Goonetilleke, D. Koutra, T. Sellis, and K. Liao, "Edge labeling schemes for graph data," in *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, pp. 1–12, 2017.
[18] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita, "Compressing graphs and indexes with recursive graph bisection," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1535–1544, 2016.
[19] Y. Liu, T. Safavi, A. Dighe, and D. Koutra, "Graph summarization methods and applications: A survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–34, 2018.
[20] M. Besta, S. Weber, L. Gianinazzi, R. Gerstenberger, A. Ivanov, Y. Oltchik, and T. Hoefler, "Slim graph: Practical lossy graph compression for approximate graph processing, storage, and analytics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–25, 2019.
[21] J. Gao, C. Zhou, J. Zhou, and J. X. Yu, "Continuous pattern detection over billion-edge graph using distributed framework," in *IEEE ICDE*, 2014.
[22] C. Wang and L. Chen, "Continuous subgraph pattern search over graph streams," in *IEEE ICDE*, 2009.
[23] C. Song, T. Ge, C. Chen, and J. Wang, "Event pattern matching over graph streams," *PVLDB*, vol. 8, no. 4, 2014.
[24] K. Vora, R. Gupta, and G. Xu, "Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations," in *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*, pp. 237–251, 2017.
[25] G. Feng, Z. Ma, D. Li, X. Zhu, Y. Cai, W. Han, and W. Chen, "Risgraph: A real-time streaming system for evolving graphs," *arXiv preprint arXiv:2004.00803*, 2020.
[26] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: taking the pulse of a fast-changing and connected world," in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 85–98, 2012.
[27] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," *ACM Transactions on Storage (TOS)*, vol. 15, no. 4, pp. 1–40, 2020.
[28] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, "faimgraph: high performance management of fully-dynamic graphs under tight memory constraints on the gpu," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 754–766, IEEE, 2018.
[29] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of streaming processing of dynamic graphs: Concepts, models, and systems," *arXiv preprint arXiv:1912.12740*, 2019.
[30] T. C. O'connell, "A survey of graph algorithms under extended streaming models of computation," in *Fundamental Problems in Computing*, pp. 455–476, Springer, 2009.
[31] C. Aggarwal and K. Subbian, "Evolutionary network analysis: A survey," *ACM Computing Surveys (CSUR)*, vol. 47, no. 1, pp. 1–36, 2014.
[32] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, "On graph problems in a semi-streaming model," *Theoretical Computer Science*, vol. 348, no. 2-3, pp. 207–216, 2005.
[33] C. Demetrescu, I. Finocchi, and A. Ribichini, "Trading off space for passes in graph streaming problems," *ACM Transactions on Algorithms (TALG)*, vol. 6, no. 1, pp. 1–17, 2009.
[34] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl, "On the streaming model augmented with a sorting primitive," in *45th Annual IEEE Symposium on Foundations of Computer Science*, pp. 540–549, IEEE, 2004.
[35] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM TOCS*, vol. 21, no. 3, pp. 270–313, 2003.
[36] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM journal on computing*, vol. 31, no. 6, pp. 1794–1813, 2002.
[37] C. Song and T. Ge, "Labeled graph sketches," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 1312–1315, IEEE, 2018.
[38] M. S. Hassan, B. Ribeiro, and W. G. Aref, "Sbg-sketch: a self-balanced sketch for labeled-graph stream summarization," in *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*, pp. 1–12, 2018.
[39] P. Zhao, C. C. Aggarwal, and M. Wang, "gsketch: on query estimation in graph streams," *PVLDB*, vol. 5, no. 3, pp. 193–204, 2011.
[40] D. E. Knuth, "Sorting and searching," 1973.
[41] P. L'Ecuyer, "Tables of linear congruential generators of different sizes and good lattice structure," *Mathematics of Computation*, vol. 68, no. 225, pp. 249–260, 1999.
[42] "Source code of gss and tcm." https://github.com/Puppy95/Graph-Stream-Sketch.
[43] L. D. Stefani, A. Epasto, M. Riondato, and E. Upfal, "Triest:counting local and global triangles in fully-dynamic streams with fixed memory size," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 825–834, 2016.
[44] S. Choudhury, L. Holder, G. Chin, K. Agarwal, and J. Feo, "A selectivity based approach to continuous pattern detection in streaming graphs," *Computer Science*, vol. 93, no. 8, pp. 939–945, 2015.

Xiangyang Gou is a Ph.D. student in the School of Electronic Engineering and Computer Science of Peking University, advised by Lei Zou. His research interests include data structures and algorithms in graph streams.

Lei Zou is a professor in Wangxuan Institute of Computer Technology of Peking University. He is also a faculty member in Big Data Center of Peking University. His research interests include graph databases and semantic data management.



Chenxingyu Zhao received the bachelor's degree from Peking University, advised by Tong Yang. He is currently a Ph.D. student at the CSE, University of Washington, advised by Arvind Krishnamurthy. He works on networking and systems, with a focus on programmable networks and disaggregated storage.

Tong Yang received the Ph.D. degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences (CAS). He is currently an Associate Professor with the Department of Computer Science, Peking University. His research interests include network measurements, sketches, IP lookups, Bloom filters, sketches, and KV stores.