Fast and Accurate Graph Stream Summarization

Xiangyang Gou, Lei Zou, Chenxingyu Zhao, Tong Yang
Peking University, China

***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 in the stream. Graph streams play important roles in cyber security, social networks, cloud troubleshooting systems and other fields. 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, like CM sketches, gSketches, TCM and gMatrix, 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 the linear space cost ($O(|E|)$, E is the edge set of the graph) and the constant update time complexity ($O(1)$) and supports *all* kinds of queries over graph streams with the controllable errors. Both theoretical analysis and experiment results confirm the superiority of our solution with regard to the time/space complexity and query results' precision compared with the state-of-the-art.**

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

I. Introduction

A. Background and Motivations

In the era of big data, data streams propose some technique challenges for existing systems. Furthermore, the traditional data stream is modeled as a sequence of *isolated* items, and the connections between the 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 the network traffic data, mining news spreading paths among the social network. In these cases the data is organized as *graph streams*. A graph stream is an unbounded sequence of items, in which each item is a vector with at least three fields (denoted by $(\langle s, d\rangle, w)$), where $\langle s, d\rangle$ represents an edge between nodes s and d, and w is the edge weight. These data items together form a dynamic graph that changes continuously and we call it *streaming graph* for convenience. Below we discuss three examples to demonstrate the usefulness of streaming graph problems.

Use case 1: *Network traffic.* The network traffic can be seen as a large dynamic 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 networks.

Use case 2: *Social networks.* In a social network, the interactions among the users can form a graph. The edges between different nodes may be 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.

Use case 3: *Troubleshooting in data centers.* Cloud systems may need to analyze communication log stream to perform real time troubleshooting. In this situation the graph stream is the sequence of communication log entries where each entry is a description of a communication from a source machine to a destination machine. In such a graph, we may perform traversal query to find out if massages created by a certain application on a source machine can reach a destination machine, or perform edge query to find the detailed information of a communication log.

These streaming graphs are very large and change fast. For example, in Twitter, there are about 100 million user login data, with 500 million tweets posted per day. For another example, in large ISP or data centers [1], there could be millions of packets every second in each link. The large volume and high dynamicity make it hard to store the graph streams efficiently with traditional data structures like adjacency lists or adjacency matrices. In the context of graph streams, there are two requirements for designing a new data structure : (1) the linear space cost (2) the constant update time. To meet these two requirements, we can either apply approximated query data structures for data streams, like the CM sketch [2], the CU sketch [3] and other sketches [4], [5], or use specialized graph summarization techniques such as gSketches [6], TCM [7] and gMatrix [8]. However, existing solutions either support limited query types or have poor query accuracy. For example, CM sketches and gSketches fail to answer queries involved with topology like reachability queries, successor queries and so on. Though TCM and gMatrix can support these queries, they have poor accuracy. More details about the related work are given in Section II. In this paper, we design a novel data structure–*G*raph *S*tream *S*ketch (GSS for short), which can support all 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-art by orders of magnitudes.

B. Our Solution

In this paper we propose GSS, which is an approximate query data structure for graph streams with linear memory usage, high update speed, high accuracy and supports all kinds of graph queries and algorithms like [9]–[11]. GSS can also be used in exiting distributed graph systems [12]–[15] Like TCM, GSS uses a hash function $H(\cdot)$ to compress the streaming graph G into a smaller graph G_h which is named a *graph sketch*. Each node v in G is mapped into a hash value $H(v)$. Nodes with the same hash value are combined into one node in G_h, and the edges connected to them are also aggregated. The compression rate can be controlled by the size of the value range of $H(\cdot)$. The higher the compression rate is, the lower the accuracy is. However, different from TCM which uses an adjacency matrix to store the graph sketch, GSS uses a novel data structure to store it. This data structure is specially designed for sparse graphs and can store a much bigger graph sketch with the same space. In the adjacency matrix, edges emitted by each node occupy one row (similar to columns). Therefore with a $m \times m$ matrix it can only represent a graph sketch with less than m nodes, which leads to high compression rate and low accuracy. However, as the graph is very sparse, most nodes have much fewer edges than m. Therefore when storing the graph sketch in GSS, we use a matrix that stores edges with different source nodes in one row, and distinguish them with fingerprints generated from their source nodes. Similarly, each column in the matrix also contains edges with different destination nodes in G_h, distinguished by another group of fingerprints generated from their destination nodes. In other words, each edge in the graph sketch is mapped to a bucket in the matrix depending on its endpoints, and marked with a fingerprint pair. If the bucket it is mapped is already occupied by other edges, we store this edge in a buffer B, which is composed of adjacency lists. With a $m \times m$ matrix we can represent a graph sketch with at most $m \times F$ nodes in GSS, where F is the size of the value range of the fingerprint (for example, a 16-bit fingerprint has $F = 65536$). With a much larger graph sketch, the accuracy is also much higher compared to TCM.

In GSS, the memory cost and update speed are greatly influenced by the size of the buffer B. As the buffer takes additional memory, and update speed in an adjacency list is linear with its size. In order to restrict its size, we propose a technique called square hashing. In this technique each edge is mapped to multiple buckets, and stored in the first empty one among them. This enlarges the chance that an edge finds an empty bucket. Besides, a few nodes in a sparse graph may still have very high degrees. If one node emits a lot of edges, these edges have high probability to evict each other when stored in one row. To solve this problem, In square hashing edges with source node v are no longer mapped to one row, but r rows, sharing memory with other source nodes. The higher degree a node has, the more buckets it may take. It is similar in the view of columns and destination nodes. This helps to ease the congestion brought by the skewness in node degrees. Experiments show that after this modification the buffer only stores less than 0.01% of the edges in the graph stream.

The key contributions of this paper are as follows:

1) We propose GSS, a novel data structure for graph stream summarization. It has small memory usage, high update speed, and supports almost all kinds of queries for graphs. Most important of all, it uses a combination of fingerprints and hash addresses to achieve very high accuracy.
2) We propose a technique called square hashing in the implementation of GSS. It helps to decrease the buffer size, improve update speed and reduce memory cost. It also eases the influence brought by the skewness in node degrees.
3) We define 3 graph query primitives and give details about how GSS supports them. Almost all algorithms for graphs can be implemented with these primitives.
4) We carry out theoretical analysis and extensive experiments to evaluate the performance of GSS, which show that when using 1/256 memory size of the state-of-the-art, our algorithm still significantly outperforms state-of-the-art for most queries.

II. Related Work

In this part we will give a brief introduction about the related works. The prior arts of graph stream summarization can be divided into two kinds. The first kind is composed of counter arrays and stores each data item in these arrays independently, ignoring the connections between them. They only support queries for edge weights, but do not support any queries involved with topology of the graph. This kind includes CM sketches [2], CU sketches [3], gSketches [6] and so on. The second kind supports all queries in the streaming graph, but suffers from poor accuracy. This kind includes TCM [7] and gMatrix [8]. In this section we introduce the CM sketch, the gSketch, TCM and gMatrix as expalmes

The CM sketch [2] is a classic data structure for data stream summarization. It can store the frequencies of different items in a data stream. It has d arrays, which are all set to 0s initially. Each array is relative with an independent hash function $h_j(x) 1 \leqslant j \leqslant d$. When inserting an item x, let $\forall j, 1 \leqslant j \leqslant d$ $count[j, h_j(x)] = count[j, h_j(x)] + 1$. Similarly, when deleting the item, let $\forall 1 \leqslant j \leqslant d$ $count[j, h_j(x)] = count[j, h_j(x)] - 1$. $count[j, h_j(x)]$ means the counter mapped by $h_j(x)$ in the j_{th} array. When querying an item x, the CM sketch carries out d hash functions and gets all the corresponding counters. Then it selects the minimum one as the frequency. The CM sketch only has overestimations, which means the query result of an item will be equal to or greater than the correct value. We can use the CM sketch to store the weights of edges in a graph stream. The memory usage is $O(|E|)$ where E is the edge set and the update needs only d memory accesses. However, it does not store the relations between edges, in other words, the topology of the streaming graph. Therefore we can not perform any topology involved queries with it. There are also some other data structures with similar functions, like CU sketches and Count sketches. They can also be applied to graph stream summarization, but also supports limited kinds of queries.

Compared to the CM sketch, gSketch [6] is specially proposed for graph streams. It aims to improve the accuracy of the CM sketch when applies to graph streams. As in CM sketches when items with small weights collide with those with large weights, in other words, mapped to the same counter, the relative error of the smaller ones will be high. The gSketch reduces their relative errors by classifying the edges with their weights and storing those with similar values together. The gSketch assumes that edges emitted by the same nodes has similar weights. It uses a small sampling of the data stream to predict of weights of edges emitted from different nodes, then divides a CM sketch into several small local sketches with different sizes, and maintains a hash table that maps the vertexes in the graph into different local sketches according to their predicted edge weights. Edges in the graph is maintained and queried in the local sketch their source vertexes are mapped to. As Only edges with similar weights will have hash collisions, the relative error of the gSketch is much smaller than the CM sketch. But like the CM sketch, it only stores the weights of edges but not any information about the topology of the graph. Therefore the gSketch also supports limited kinds of queries

TCM [7] is the state-of -the-art of data structures for graph stream summarization. It is composed of an adjacency matrix that stores the compression of the streaming graph. 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 weights of an edge in G_h is an aggregation of the weights of all edges mapped to it. A hash table that stores the hash value and the original ID pairs $\langle H(v), v\rangle$ is built in this map procedure to retrieve the original node IDs for some queries. Then TCM uses an adjacency matrix to represent the graph sketch. If we represent the size of the value range of $H(\cdot)$ with M, we need to build an $M \times M$ adjacency matrix. When the memory is sufficient, we can also build multiple sketches with different hash functions, and report the most accurate value in query.

In order to satisfy the demand on memory usage, the size of the adjacency matrix, $M \times M$ has to be within $O(|E|)$, which means $M \ll |V|$ for a sparse streaming graph where $\frac{|E|}{|V|}$ is usually within 10. This means the graph sketch G_h is usually much smaller than G, a lot of nodes and edges will be aggregated. As a result, the accuracy of TCM is poor.

The gMatrix [8] is a variant of TCM. Its structure is similar to TCM. But it uses reversible hash functions to generate graph sketches in order to omit the memory and time cost of using the hash table which stores the node IDs. It also extends TCM to more queries like edge heavy hitters and so on. However, different from the accurate hash tables, the reversible hash function introduces additional errors in the reverse procedure. Therefore the accuracy of gMatrix is no better than TCM, sometimes even worse.

There are some graph algorithms for statistic graph compression [16]–[18] or specific queries over graph streams [19]–[21]. However, they are either not suitable for high dynamic graph streams or too limited in functions. We do not introduce them in detail due to space limit.

III. Problem Definition

Definition 1: **Graph Stream**: A graph stream is an unbounded timing 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 dynamic 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 edge in the streaming graph G is SUM of all edge weights sharing the same endpoints. The weight w can be either positive or negative. When $w < 0$, it means deleting a former data item.

Example 1: A sample graph stream S and the corresponding streaming graph G are both shown in Fig. III. Each node has an ID that uniquely identifies itself. If an edge appears multiple times, its weights are 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, but the time cost of inserting an edge is $O(|V|)$, which is unacceptable due to the high speed of the graph stream.

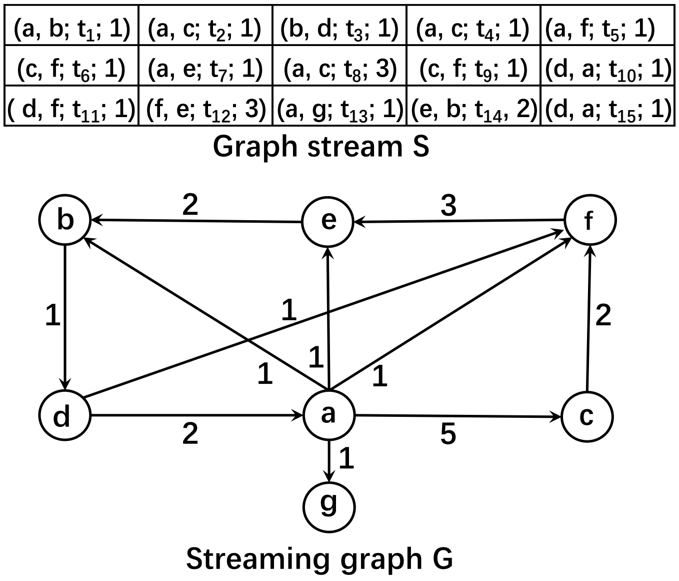
The goal of our study is to design a *linear* space cost data structure with efficient query and update algorithms over high speed graph streams. To meet that goal, we allow some approximate query results but with small and controllable errors. However, traditional graph stream summarization approaches either cannot answer graph topology queries such as reachability queries (such as CM sketches [2] and gSketches [6]) or fail to provide accurate query results (such as TCM [7] and gMatrix [8]). Therefore, in this paper, we design a novel graph stream summarization strategy.

In order to give a definition of the graph stream summarization problem, First we define the *graph sketch* as follows:

Definition 2: **Graph Sketch**: a graph sketch of $G = (V, E)$ is a samller graph $G_h = (V_h, E_h)$ where $|V_h| \leqslant |V|$ and $|E_h| \leqslant |E|$. A map function $H(\cdot)$ is used to map each node in V to a node in V_h, and edge $e = \overrightarrow{s, d}$ in E is mapped to edge $\overrightarrow{H(s), H(d)}$ in E_h. The weight of an edge in E_h is the SUM of the weights of all edges mapped to it.
Formally, we define our *graph stream summarization* problem as follows.

Definition 3: **Graph Stream Summarization**: Given a streaming graph $G = (V, E)$, the *graph stream summarization* problem is to design a graph sketch $G_h = (V_h, E_h)$, and the

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



(a, b; t_1; 1)	(a, c; t_2; 1)	(b, d; t_3; 1)	(a, c; t_4; 1)	(a, f; t_5; 1)
(c, f; t_6; 1)	(a, e; t_7; 1)	(a, c; t_8; 3)	(c, f; t_9; 1)	(d, a; t_{10}; 1)
(d, f; t_{11}; 1)	(f, e; t_{12}; 3)	(a, g; t_{13}; 1)	(e, b; t_{14}, 2)	(d, a; t_{15}; 1)

Fig. 1. A sample graph stream

corresponding data structure DS to represent G_h, where the following conditions hold:

1) There is a function $H(\cdot)$ that map nodes in V to nodes in V_h;
2) The space cost of DS is $O(|E|)$;
3) DS changes with each new arriving data item in the streaming graph and the time complexity of updating DS should be $O(1)$;
4) DS supports answering any query over the original streaming graph G with small and controllable errors.

In the context of streaming graphs, G changes with every data item in the graph stream S, which is mapped to updating the graph sketch G_h, and conducted in data structure DS. For every new item $(\overrightarrow{s,d};t;w)$ in S, we map edge $\overrightarrow{s,d}$ in G to edge $\overrightarrow{H(s),H(d)}$ in G_h with weight w and then insert it into G_h. There are three cases for inserting edge $\overrightarrow{H(s),H(d)}$ into G_h. First, edge $\overrightarrow{H(s),H(d)}$ has been in the graph sketch G_h already, we only update the edge weight. Second, if both nodes $H(s)$ and $H(d)$ have been in the graph sketch G_h but edge $\overrightarrow{H(s),H(d)}$ does not occur, we introduce edge $\overrightarrow{H(s),H(d)}$ with the weight w to G_h directly. Third, if either of $H(s)$ or $H(d)$ does not occur in $V(G_h)$, we introduce the node and then add the edge $\overrightarrow{H(s),H(d)}$ with the weight w into G_h.

Similarly, queries over G are also mapped to the same kind of queries over the graph sketch G_h. In order to support any kind of graph queries, we first define three graph query primitives as follows, since many kinds of graph queries can be answered using these primitives.

Definition 4: **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.

When answering queries over the streaming graph G, we return the answer of the same queries in the graph sketch G_h as an approximate answer. Therefore, the data structure DS should be able to conduct all three query primitives over G_h to support various kinds of queries.

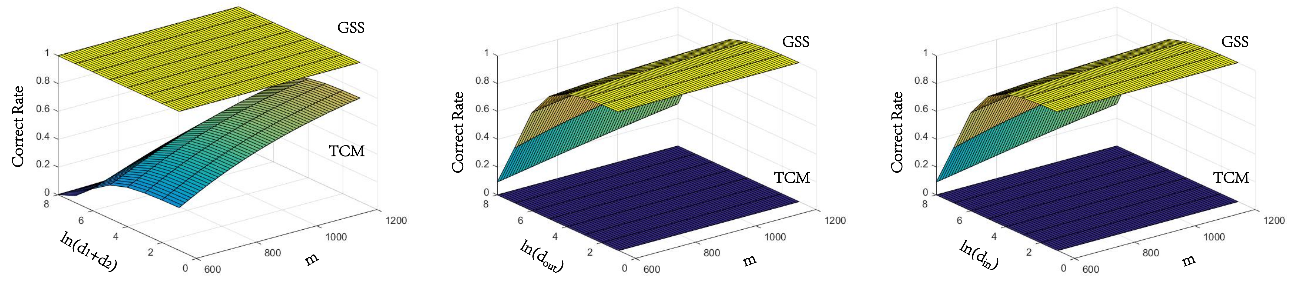
Although TCM [7] and gMatrix [8] can answer these query primitives, they suffer from inaccuracy of query results. Figure 2 illustrates the accuracy rate of different approaches for these graph query primitives. For the simplicity of the presentation, we fix the graph size $|E| = 421578$ and $|V| = 34546$, which is the size of a graph dataset we use in the experiment. Since TCM, gMatrix, and our GSS all use the *matrix* as the storage data structure, thus, we use m to represent to side length of the matrix. In the edge query primitive, d_1 and d_2 represent the out-degree of the source node and the in-degree of the destination node, respectively. In the 1-hop successor query primitive, d_{out} represents the out degree of the node we query, and in the 1-hop precursor query primitive, d_{in} represents the in degree of the node we query. From Figure 2, we know that our method outperforms TCM and gMatrix by orders of magnitudes in the accuracy ratio, especially in the 1-hop successor/precursor queries. Detailed theoretical analyses are given in Section V-C. Note that the theoretical results are consistent with our experimental results in section VI, which confirms the superiority of our approach in the query accuracy.

IV. GSS: Basic Version

In this section, we describe a conceptually simple scheme to help illustrate intuition and benefit of our approach. The full approach, presented in Section V, is designed with more optimizations. As stated above, to produce a graph stream summarization, we first need to design a graph sketch $G_h = (V_h, E_h)$ for the streaming graph G. Initially, we use the same strategy as TCM to generate the graph sketch. We choose a hash function $H(\cdot)$ with value range $[0, M)$, then G_h is generated as following:

1) **Initialization**: Initially, $V_h = \varnothing$, and $E_h = \varnothing$.
2) **Edge Insertion**: For each edge $e = (s,d)$ in E with weight w, we compute hash values $H(s)$ and $H(d)$. If either node with ID $H(s)$ 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 hash tables to make this mapping procedure reversible. This needs $O|V|$ additional memory, as $|V| \leqslant |E|$, the overall memory requirement is still within $O(|E|)$.



(a) Edge Query

(b) 1-hop Successor Query

(c) 1-hop Precursor Query

Fig. 2. Theoretical Accuracy

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

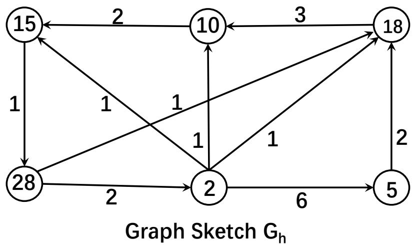


Fig. 3. A sample map function

Example 2: A graph sketch G_h for the streaming graph G in Figure III is shown in Figure 6. 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 $(2, 5)$ is 6, which is the summary of the weight of edge (a, c) and edge (a, g) in G.

Obviously, the size of the value range of the map function $H(\cdot)$, which we represent with M, will significantly influence the accuracy of the summarization, especially in the 1-hop successor / precursor query primitives, as shown in Figure 4 (In the figure of the edge query, d_1 and d_2 means the in-degree of the source node and the out-degree of the destination node of the queried edge. In the figure of the 1-hop successor / precursor query, d_{in} and d_{out} means the in-degree and the out-degree of the queried node, respectively). In a uniform mapping with the hash function, each node in G has the probability $\frac{1}{M}$ to collide with another, which means they are mapped to the same node in G_h. In the 1-hop successor / precursor queries, this kind of node collisions will definitely lead to errors. Figure 4 shows that, only when $\frac{M}{|V|} > 200$, the accuracy ratio is larger than 80% in 1-hop successor/precursor queries. When $\frac{M}{|V|} \leq 1$, the accuracy ratio falls down to nearly 0, which is totally unacceptable.

Both TCM and the gMatrix resort to an adjacency matrix to represent G_h. In this case, the matrix rank m equals to M, i.e, the value range of the map function. To keep the memory usage of the graph sketch within $O(|E|)$ (Condition 2 in Definition 3), m must be less than $\sqrt{E}$, that means $m = M < \sqrt{E} \ll |V|$ for a sparse streaming graph. According to our theoretical analysis[2] in Figure 4, the query results' accuracy is quite low in them. Our experiments in Section VI also confirm the theoretical analysis.

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

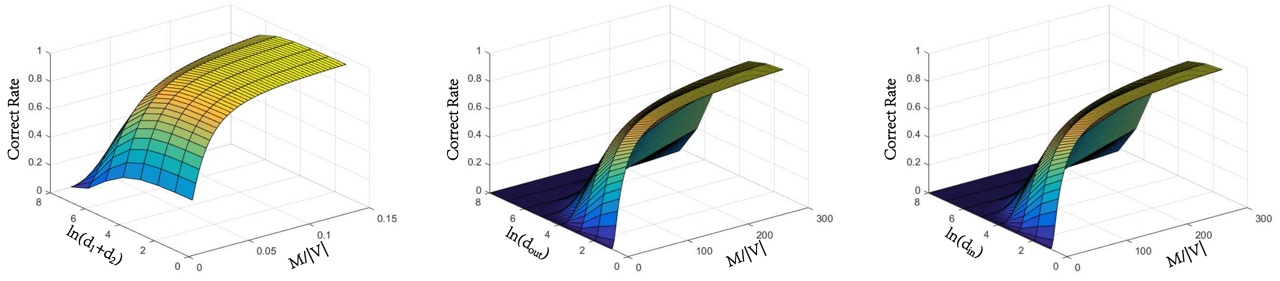
Definition 5: **GSS**: Given a streaming graph $G = (V, E)$, we have a hash function $H(\cdot)$ with value range $[0, M)$ to map each node v in graph G to node $H(v)$ in graph sketch G_h. Then we use the following data structure to represent the graph sketch G_h:

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 $h(v)(0 \leqslant h(v) \leqslant m)$ and a fingerprint $f(v)(0 \leqslant f(v) \leqslant F)$ 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 $[\langle f(s), f(d) \rangle, w]$ 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 other previous inserted edges already.

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 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 6 in shown in Figure 5. 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.

[2] The detailed analyses are given in Section V-C2

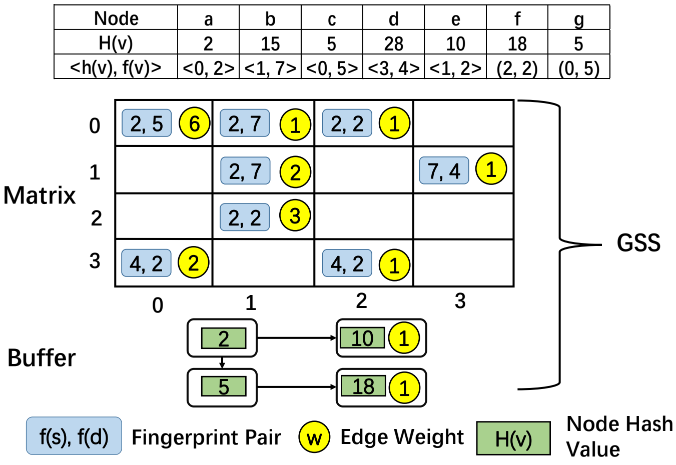


(a) Edge Query

(b) 1-hop Successor Query

(c) 1-hop Precursor Query

Fig. 4. Influence of M on Accuracy



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)

	0	1	2	3
0	2, 5 ⑥	2, 7 ①	2, 2 ①	
1		2, 7 ②		7, 4 ①
2		2, 2 ③		
3	4, 2 ②		4, 2 ①	

Fig. 5. A sample of the basic version of data structure

We discuss the insertion and primitive query operations over GSS as follows:

Edge Updating: When a new item $(s, d; t; w)$ comes in the graph stream S, we map it to an edge $\overrightarrow{H(s), H(d)}$ with weight w in graph sketch G_h. Then we find the bucket in row $h(s)$, column $h(d)$. If the bucket is empty, we store the fingerprints pair $\langle f(s), f(d)\rangle$ together with the edge weight w in the bucket. If it is not empty, we compare the fingerprint pair of this edge with the fingerprint pair $\langle f(s'), f(d')\rangle$ that is in the bucket already. If they are same, we add the weight w to the existing one; otherwise, it means this bucket has been occupied by other edges, and we store edge $\overrightarrow{H(s), H(d)}$ in the adjacency list in the buffer B. We call this kind of edges as *left-over* edges.

Graph Query Primitives: The three primitives (defined in Definition 4) are all supported with our proposed data structure GSS.

Edge Query: Given an edge query $e = \overrightarrow{s, d}$, we work as follows. We check the bucket in row $h(s)$, column $h(d)$ in the matrix. Let $\langle f(s'), f(d')\rangle$ be the fingerprint pair stored at the bucket. If $\langle f(s'), f(d')\rangle$ equals to the the fingerprint pair $\langle f(s), f(d)\rangle$ of edge $\overrightarrow{s, d}$, we return the weight in the bucket. Otherwise we search the buffer B for edge $\overrightarrow{H(s), H(d)}$ using the adjacency list. If we cannot find it in the matrix X or in the buffer B, we return -1, i.e. reporting that the edge $e = \overrightarrow{s, d}$ does not exists.

1-*hop Successor Query*: To find the 1-hop successors of node v, we work as follows. First, we search all buckets in row $h(v)$ of the matrix X. If a bucket in row $h(v)$ and column c has a fingerprint pair $\langle f(v), f(v_s)\rangle$, we add node $H(v_s) = c \times F + f(v_s)$ to the 1-hop successors set SS. After that, we also need to search the buffer area to find all edges with source node $H(v)$, and add its destination node to the 1-hop successors set S. We return -1 if we find no result, i.e., $|SS| = 0$. Otherwise, for each $H(s)$ in successors set SS, we obtain the original node IDs by accessing the hash table $\langle v, H(v)\rangle$.

1-*hop Precursor Query*: To find the 1-hop precursors of node v, 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.

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 distinguish them. For example, edge $\overrightarrow{15, 28}$ and edge $\overrightarrow{10, 15}$ are all stored in row 1, but they have different source node fingerprints, namely 2 and 7, thus we know exactly which nodes they are from. It is similar in columns. 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 size of matrix as TCM. With a 4×4 matrix as in Figure 6, TCM can only support a map function with $M = 4$, and the number of nodes in the graph sketch will be no more than 4, thus the accuracy will he much poorer.

V. GSS: Augmented Algorithm

A. Data Structure

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 the linear time $O(|B|)$ if the edge must goto the buffer B, where $|B|$ represents the number of all left-over edges. Therefore $|B|$ both influences the memory and time cost. In this section, we design several solutions to reduce the size of buffer B. In the basic version, an edge is pushed into buffer B if and only if its hash address in the matrix X has been occupied. Consider a high degree vertex 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 5), 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. As we know, the real-world graphs' vertex degrees usually follow the power law distribution. In other words, there are a few crowded rows, and they lead to most left-over edges in buffer B. Actually, 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

$$\{(h_i(s), h_j(d))|(1 \leqslant i \leqslant r, 1 \leqslant j \leqslant r)\}$$

where $h_i(s)$ is the row index and $h_j(d)$ 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.

Example 4: An example of square hashing is shown in Figure V. The inserted edge is mapped to 9 buckets, and the first 2 with address $(h_1(s), h_1(d))$ and $(h_1(s), h_2(d))$ have been already occupied. Therefore the edge is inserted in the third mapped bucket. In the bucket, we store the weight, the fingerprint pair, together with an index pair $\langle 1, 3 \rangle$ which indicates the position of this bucket in the mapped buckets sequence. We will talk about the use of index pair later.

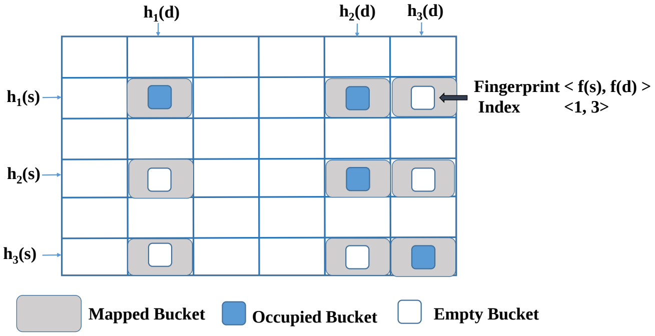


Fig. 6. The square hashing

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 P to represent the probability that $\forall 1 \leq i \leq r, h_i(v_1) = h_i(v_2)$. Then we have $P = \prod_{i=1}^{r} Pr(h_i(v_1) = h_i(v_2))$. 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 of indexing 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* [22] 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

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

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

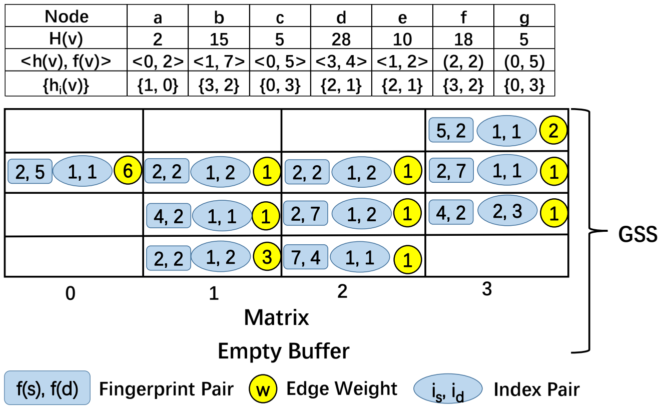
$$\{h_i(v)|h_i(v) = (h(v) + q_i(v))\%m, 1 \leqslant i \leqslant r\} \tag{2}$$

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 (i_s, i_d), supposing that the bucket that contains this room has an address $(h_{i_s}(s), h_{i_d}(d))$. 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 [8]. 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 the 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 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 $(f(s), f(d))$ and the index pair (i_s, i_d) 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.

Example 5: An example of the modified version is shown in Fig. V-A. In the matrix we stored G_h in Fig. 6, which is a compressed graph of G in Fig.III. In this example we set $F = 8$, $m = 4$, $r = 2$, and the equation in the linger congruence method is

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



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)
{h_i(v)}	{1, 0}	{3, 2}	{0, 3}	{2, 1}	{2, 1}	{3, 2}	{0, 3}

	0	1	2	3
0				5, 2 1, 1 2
1	2, 5 1, 1 6	2, 2 1, 2 1	2, 2 1, 2 1	2, 7 1, 1 1
2		4, 2 1, 1 1	2, 7 1, 2 1	4, 2 2, 3 1
3		2, 2 1, 2 3	7, 4 1, 1 1	

Fig. 7. A sample of the modified version of data structure

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.

In the following, we illustrate the four basic operators in this data structure GSS.

Edge Updating: When a new item $(s, d, t; w)$ 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 $\{(h_i(s), h_j(d)) | 1 \leqslant i \leqslant r, 1 \leqslant j \leqslant r\}$ 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 $(f(s), f(d))$ and the index pair (i_s, i_d) and weights w in it, and end the procedure. If it is not empty, we check the fingerprint pair $(f(s'), f(d'))$ and the index pair (i'_s, i'_d) stored in the bucket. If the fingerprint pair and the index pair are all equal to the corresponding pairs of the new inserted edge $\overrightarrow{H(s), H(d)}$, we add w to the weights in it, and end the procedure. Otherwise it means this bucket has been occupied by other edges 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, like the basic version of GSS.

Graph Query Primitives: The three graph query primitives are supported in the modified data structure 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 the same square hashingmethod 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 $(f(s), f(d))$ and the index pair (i_s, i_d), we return its weight as the result. If we find no results in the r^2 buckets, we search the buffer for edge $\overrightarrow{H(s), H(d)}$ and return its weights. If we still can not find it, we return -1.

1-hop Successor Query: to find the 1-hop successors of node v, we map it to node $H(v)$ in G_h. Then we compute its hash address sequence $h_i(v)$ 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 $((f(v), f(x))$ and index pair (i_s, i_d) where $f(x)$ is any integer in range $[0, F)$ and i_d is any integer in range $[1, r]$, we use $f(x)$, i_d and C to compute $H(x)$ as stated above. Then we add $H(x)$ 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 node to SS. We return -1 if we find no result, otherwise we obtain the original node IDs from SS by accessing the hash table $\langle H(v), v \rangle$.

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 the r different columns. These rows or columns are shared by the edges with different source nodes or destination nodes. The higher degree a node has, the more buckets its edges may take. This eases the congestion brought by skewness in node degrees. Moreover, as each bucket has multiple mapped buckets, it has higher probability to find an empty one. Obviously, square hashing will reduce the number of *left-over edges*.

B. Further Improvements

There are some other improvements which can be implemented to GSS. They can further improve the speed in some situations.

1) Mapped Buckets Sampling: In the modified version of GSS, each edge has r^2 mapped buckets. We usually set r to integers from 4 to 8. When the skewness of node degrees is serious, r can be larger. If we check all the r^2 buckets when inserting an edge, it will be time consuming. To improve the updating speed, which is very important for graph stream summarization, 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 query, 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. We add the fingerprint of the source node and the destination node of e to get a seed $seed(e)$, then we compute a k length sequence as

$$\begin{cases} q_1(e) = (a \times seed(e) + b)\%p \\ q_i(e) = (a \times q_{i-1}(e) + b)\%p, (2 \leqslant i \leqslant r) \end{cases} \quad (4)$$

where a, b and p are the same integers used above. We choose the k buckets with address

$$\left\{ (h_{\left\lfloor \frac{q_i(e)}{r} \right\rfloor \% r}(s), h_{(q_i(e)\% r)}(d)) | 1 \leqslant i \leqslant k \right\} \quad (5)$$

$\{h_i(s)\}$ and $\{h_i(d)\}$ are the hash address sequence of the source node and the destination node, respectively.

2) Multiple Rooms: When the memory is sufficient, we do not need to use multiple matrices to increase accuracy as TCM, as the accuracy is already very high. Instead, in order to further decrease the buffer size, we can separate each bucket in the matrix into l rooms, and each room contains an edge. When performing the basic operators, we use the same process as above the find the buckets we need to check, and search all the rooms in them to find qualified edges or empty rooms.

However, when the rooms in each bucket are stored separately, the speed will probably decrease, as we can not fetch the l rooms in one memory access in most cases, and multiple memory accesses increase the time cost. As shown in Fig. V-B2, we separate the bucket into 3 area: the index area, the fingerprint area, and the weight area. Each area contains the corresponding parts of the l rooms. When we check this bucket to find certain edges, we can first check all the index pairs. If we find a matched index pair, we check the corresponding fingerprint pair, and if the fingerprint pair is also matched, we fetch the corresponding weights. If we do not find any matched index pair, we can just move on and do not need to check the fingerprint pairs any more. As the index pairs are very small, usually no more than 1 byte, we can fetch all the index pairs in one memory access. This will omit a lot of unnecessary memory accesses.

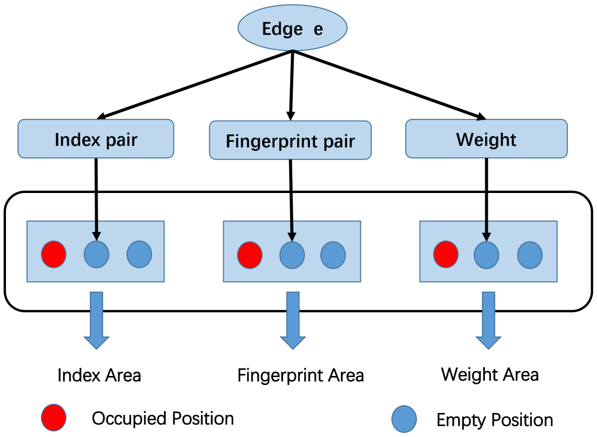


Fig. 8. Bucket Separation

C. Analysis

1) Memory and Time Cost Analysis: As stated above, The memory cost of GSS is $O(|E|)$. In implementations we usually set the size of the matrix to be near to $|E|$. In this case the buffer stores less than 0.01% of the edges, which will be shown in section V-C3 and VI-G. Therefore its memory cost is negligible. As for the hash table which stores the original node ID and the corresponding $H(v)$, because its memory cost is $O(|V|)$ where $|V| < |E|$, the total memory is still $O(|E|)$.

The average time cost in update in $O(1)$. In the matrix, the update need constant time. As very few (less than 0.01%), even none edges are stored in the buffer, we do not need to access the buffer in most cases. Even when we need to check it, the speed is still high as there are few edges in it. Therefore the average time cost of inserting an edge is still $O(1)$. As for the time cost of query, it depends on the kind of query and the algorithm we use. We consider the time cost of the operators as an evaluation of the query speed. For edge query operator, as the process is the similar to edge insertion, the time cost is also $O(k)$. For the 1-hop successor query and the 1-hop precursor query, the time cost is $O(r \times m)$, where r is the length of the hash address sequence, and m is the side length of the matrix.

2) Accuracy Analysis: In this part we evaluate the accuracy of GSS. Before we analyze the probability of errors, we first propose the following theorem:

Theorem 1: The storage of the graph sketch G_h in the data structure of GSS is accurate. Which means for any edge $e_1 = \overrightarrow{H(s_1), H(d_1)}$ and $e_2 = \overrightarrow{H(s_2), H(d_2)}$ in G_h, the weights of them will be added up if and only if $H(s_1) = H(s_2), H(d_1) = H(d_2)$.

The prove of the theorem can be referred in the full version of the paper. Here we skip it because of space limit. This theorem means we only need to consider the procedure of mapping G to G_h, as all errors happen in this procedure. In the mapping procedure, multiple nodes in G may be mapped to one node in G_h, and multiple edges in G may be mapped to one edge in G_h. This introduces over-estimations in queries, as the result we get is an aggregation of multiple items. The probability the one node v_1 collides with another node v_2 is obviously $\frac{1}{m \times F}$ in GSS. We use $\hat{P}$ to represent the probability of the following event:

Definition 6: **Edge Collision**: An edge collision means that given an edge e, there is at least one e' in G and $e' \neq e$ which satisfies $H(e) = H(e')$ in the compressed graph G_h.
We set $P = 1 - \hat{P}$, and P is the main component of the error rate of all the 3 graph query primitives. In the 1-hop successor query query for a node v, We will get a correct answer if and only if for each v' in G which is not a 1-hop successor of v, edge (v, v') do not collide with any edge in G, and the probability of such an event is P. Therefore the correct rate of 1-hop successor query is $P^{|V|-d}$, where $|V|$ is the number of nodes in G, and d is the number of 1-hop successors of the queried node v. It is the same in the 1-hop precursor query. In the edge query, P is just the correct rate. Therefore we need to compute P to evaluate the accuracy of GSS.

For an edge $e = \overrightarrow{s, d}$ in G, we assume there are D edges with source node s or destination node d in G besides e, and there are totally $|E|$ edges in G. We represent the size of the value range of the map function $H(\cdot)$ with M. For an edge share no common endpoints with e , the probability that it collides with e in map function $H(\cdot)$ is:

$$p_1 = \frac{1}{M^2} \tag{6}$$

The probability that all the $|E| - D$ edges have no collisions with e is

$$Pr_1 = (1 - p_1)^{|E|-D} \tag{7}$$

For those D edges connected to e, as one of the two end points is the same, the probability that such an edge has a collision with e is

$$p_2 = \frac{1}{M} \tag{8}$$

The probability that all the D edges have no collisions with e is

$$Pr_2 = (1 - p_2)^D \tag{9}$$

Therefore the correct rate of e, in other words, all the $|E|$ edges do not have collisions with e in mapping is

$$\begin{aligned} P &= Pr_1 \times Pr_2 \\ &= ((1-p_1)^{|E|-D}) \times ((1-p_2)^D) \\ &= e^{-p_1 \times (|E|-D)} \times e^{-p_2 \times D} \\ &= e^{-\frac{|E|-D}{M^2}} \times e^{-\frac{D}{M}} \\ &= e^{-\frac{|E|+(M-1)\times D}{M^2}} \end{aligned} \tag{10}$$

In GSS we have $M = m \times F$, where m is the length of the matrix, and F is the maximum size of the fingerprints. The above correct rate is usually very high in applications. For example, suppose that the fingerprints we use are 8-bit, in other words, $F = 256$, and when querying an edge e, we have $|E| = 5 \times 10^5$,$D = 200$. We use a matrix with side length $m = 1000$. Then the correct rate of this edge query is $e^{-0.00078} = 0.9992$. On the other hand, in TCM the accuracy analysis is the same as GSS but we have $M = m$. With the same matrix size, TCM only has a probability of 0.497 to get a correct weight for e in above situation.

3) Buffer Size Analysis: After all the improvements, the buffer in GSSis 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. Assuming that there are already N different edges in the graph stream, and among them D edges have common source node or common destination node with e. The length 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 $N-D$ non-adjacent edges 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 are randomly inserted in an area of $r \times m$ (r rows or r columns in the matrix), 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 probability is rather small. For example if $N = 1 \times 10^6$, $D = 10^4$, we still set the side length of the matrix to $w = 1000$, and set $r = 8$, $l = 3$, $k = 8$, the upper bound probability of insertion failure is only 0.002. Experiments show that when the size of matrix is nearly equal to the number of edges, there will be almost no edges insert into the buffer. The details of the experiments will be shown in the next section.

VI. Experimental Evaluation

In this section, we show our experimental studies of GSS. Among the prior arts, only TCM and the gMatrix support all kinds of queries, and despite of its versatility, the gMatrix has the same accuracy as TCM. Therefore we regard TCM as state of the art and compare GSS with it in experiments. We first explain the data sets in experiments (VI-A) and evaluation metrics (VI-B). Next, we compare GSS with prior arts on the three graph query primitives: edge query , 1-hop successor query, 1-hop precursor query (VI-D) and two compound queries, node queries (VI-E) and reachability queries (VI-F). We define a node query for v as computing the summary of the weights of all edges with source node v, and define reachability query as finding out if we can travel to node d from node s along the edges in the graph. We also evaluate the size of buffer (V-C3)and update speed of GSS (I). Finally, we summarize our experimental studies (VI-I).

All experiments are performed on a server with dual 6-core CPUs (Intel Xeon CPU E5-2620 @2.0 GHz, 24 threads) and 62 GB DRAM memory, running Ubuntu. All algorithms including GSS and TCM are implemented in C++.

A. Data Sets

We choose three real world data sets from Stanford Network Analysis Platform (SNAP)[3]. These data sets are all unweighted, directed real word graphs. We use the Zipfian distribution to generate weights for the edges in the data set, and insert the edges into the data structure one by one to simulate the procedure of real-world incremental updating. Details of three data sets are described as follows:

1)**email-EuAll**[4].This data set is communication network data generated using email data from a large European research institution for a period of 18 months. Each node in the directed graph corresponds to an email address. Each edge between node s and d represents that s sent at least one email to d. The directed graph contains 265214 nodes and 420045 edges. 2)**web-NotreDame**[5].The second data set is a web graph collected from the University of Notre Dame. Nodes represent web pages and directed edges represent hyperlinks between pages. The directed graph contains 325729 nodes and 1497134 edges. 3)**cit-HepPh**[6].It is the Arxiv HEP-PH (high energy physics phenomenology) citation graph. If a paper s cites paper d, the data set contains a directed edge from s to d. The data set covers 34,546 papers as nodes with 421,578 edges.

The function we use to cumulate the edge weights is addition. In this case, TCM and GSS only have over-estimations.

B. Metrics

In this part we give a definition of the metrics we use in experiments.

Average Relative Error (ARE): ARE measures the accuracy of the reported weights in edge queries and node queries. Given a query q, the *relative error* is defined as

$$RE(q) = \frac{\hat{f(q)}}{f(q)} - 1$$

where $f(q)$ and $\hat{f(q)}$ are the real answer and the estimated value of q. When giving a query set $Q = \{q_1, q_2, ..., q_k\}$, the *average relative error* (ARE) is measured by averaging the relative errors over all queries of Q. A more accuracy data structure will have smaller ARE for such a query set Q.

Average Precision We use average precision as the evaluation metric in 1-hop successor queries and 1-hop precursor queries. Given such a query q, we use SS to represent the accurate set of 1-hop successors / precursors of the queried node v, 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 define the precision of q as:

$$Precision(q) = \frac{|SS|}{|\hat{SS}|}$$

Average precision of a query set Q is the average value of the precision of all queries in Q. A more accuracy data structure will have higher *Average Precision* for such a query set Q

True Negative Recall: It measures the accuracy of the reachability query. Because connectives of all edges are kept, there is no false negatives in TCM and GSS, which means if we can travel to d from s in the streaming graph, the query result of these data structures will be definitely yes. Therefore in experiments we use reachability query sets $Q = \{q_1, q_2, ..., q_k\}$ where $\forall q_i \in Q$, source node s and destination node d in q_i are unreachable. *True negative recall* is defined as the number of queries reported as unreachable divided by the number of all queries in Q.

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

C. 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 *fsize* to represent the fingerprint size by short. We apply all improvements to GSS, and the parameters are as follows. Each bucket in the matrix contains $l = 4$ rooms. The length of the address sequences is $r = 8$, and the number of *candidate buckets* for each edge is $k = 8$. As for TCM, we apply 4 graph sketches to improve its accuracy, and allow it to use larger memory, because otherwise the gap between it and GSS will be so huge that we can hardly compare them in one figure. In edge query primitives, we allow TCM to use 8 times memory, and in other queries we implement it with 256 times memory, as its accuracy is too poor in these queries. This ratio is the memory used by all the 4 sketches in TCM divided by the memory used by GSS with 16 bit fingerprint. When the size of GSS varies, the size of matrix in TCM also varies correspondingly to keep the ratio unchanged.

D. Experiments on query primitives

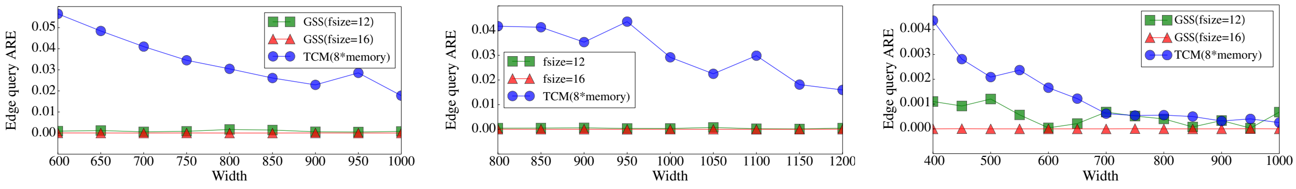
In this section, we evaluate the performance of GSS in the 3 basic graph query primitives: the edge query, the 1-hop precursor query and the 1-hop successor query. We implement TCM with 8 times of memory in edge queries, and 256 times of memory in other 2 query primitives to make the result comparable. Figure 9, Figure 10, and Figure 11 show the ARE of edge queries and average precision of 1-hop precursor / successor queries for the data sets email-EuAll, web-NotreDame and cit-HepPh, respectively. To reduce random error introduced by the selection of the data sample, 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 both GSS and TCM, the ARE decreases, and the precision

[3] http://snap.stanford.edu/data

[4] http://snap.stanford.edu/data/email-EuAll.html

[5] http://snap.stanford.edu/data/web-NotreDame.html

[6] http://snap.stanford.edu/data/email-EuAll.html

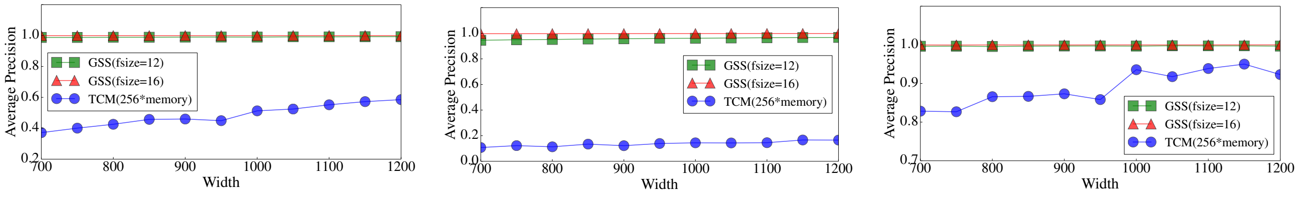


(a) email-EuAll

(b) web-NotreDame

(c) cit-HepPh

Fig. 9. Average Relative Error of Edge Queries

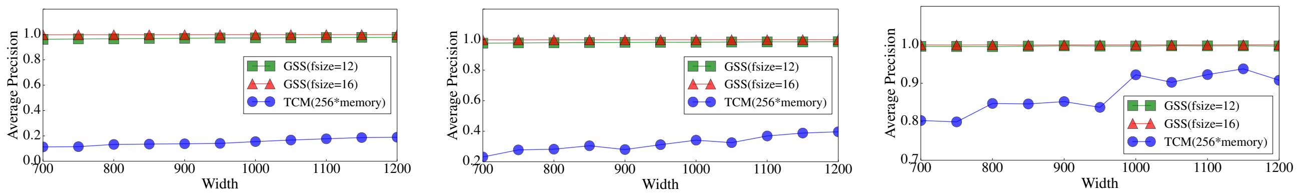


(a) email-EuAll

(b) web-NotreDame

(c) cit-HepPh

Fig. 10. Average Precision of 1-hop Precursor Queries



(a) email-EuAll

(b) web-NotreDame

(c) cit-HepPh

Fig. 11. Average Precision of 1-hop Successor Queries

increases with the growth of the width of the matrix. This trend is not very significant in GSS as the accuracy is high and there are no errors in most experiments. Also, when the length of fingerprint becomes longer, the accuracy of GSS increases. The gap between TCM and GSS is smaller in cit-HepPhas the graph is smaller compared to other datasets, but it should be noted that this result is based on the condition that TCM consumes much more memory. The accuracy of GSS still out-performs TCM a lot in this data set.

E. *Experiments on Node Query*

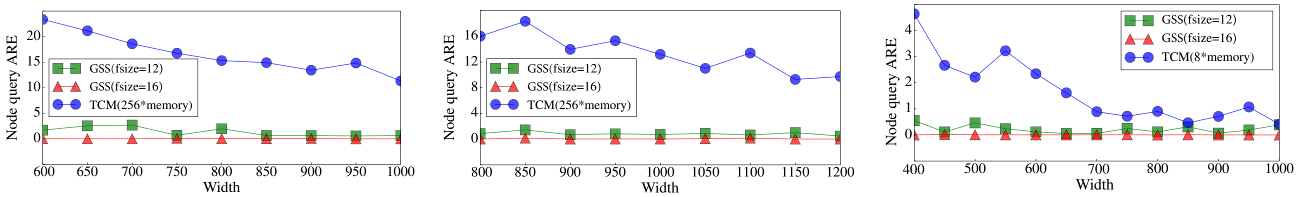
In this section, we evaluate the performance of GSS in estimating the accuracy of node query. A node query for a node v is to compute the summary of the weights of all edges with source node v. For each dataset, node query set contains all nodes in the graph stream. Figure 12 shows the ARE of node queries in data sets email-EuAll, web-NotreDame and cit-HepPh, respectively. As TCM has very poor accuracy in topology queries, to make TCM comparable with GSS, we fixed the memory of TCM 256 times as large as GSS. The figure shows that although we unfairly fix the ratio of memory used by TCM and GSS, GSS still can achieve better performance than TCM.

F. *Experiments on Reachability Query*

In this section, we evaluate the performance of GSS in supporting reachability queries. Each reachability query set Q contains 100 unreachable pairs of nodes which are randomly generated from the graph. Figure 13 shows the true negative recall of reachability query for the data sets email-EuAll, web-NotreDame and cit-HepPh, respectively. From the figures we can see that the accuracy of GSS is much higher than TCM even when TCM uses much larger memory. The gap varies with the size of the graph. Along with increasing the memory and the length of the fingerprint, GSS can achieve better performance. From the figures we can also see that the accuracy of TCM is so poor in this query that it can barely support it.

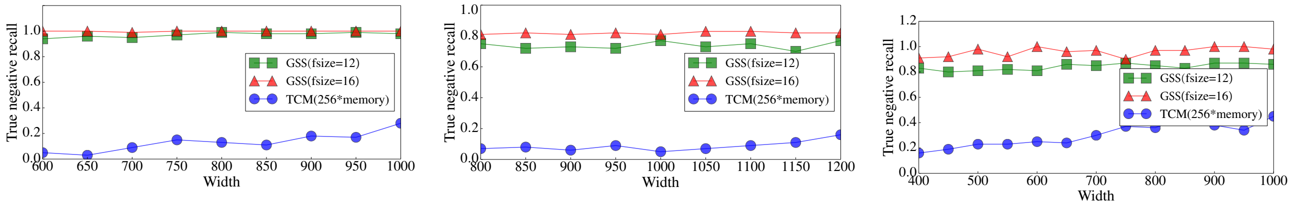
G. *Experiments on Buffer Size*

In this section, we evaluate the buffer size of GSS. Figure 14 shows the buffer percentage for three data sets email-EuAll, web-NotreDame and cit-HepPh, respectively. The above results show that the buffer percentage for three data sets is below 10^{-4}. The buffer size decreases with the increment of the size of the matrix, and becomes 0 in most experiments when the matrix size is close to $|E|$. In this case, the overhead brought by the insertion failure in the matrix is nearly 0. Notice



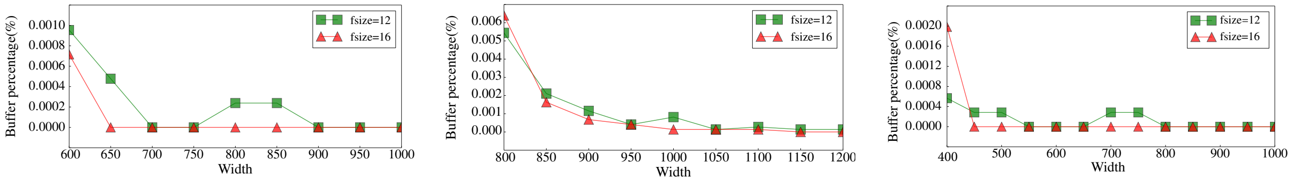
(a) email-EuAll (b) web-NotreDame (c) cit-HepPh

Fig. 12. Average Relative Error of Node Queries



(a) email-EuAll (b) web-NotreDame (c) cit-HepPh

Fig. 13. True Negative Recall of Reachability Queries



(a) email-EuAll (b) web-NotreDame (c) cit-HepPh

Fig. 14. Buffer Percentage

that there may be some disturbance in the decreasing trend of the kinked line because of the randomicity of the hash function, as the buffer is so small that the influence of several edges will be noticeable.

H. Experiment on update speed

In this section we evaluate the update speed of GSS. We compare the update speed of GSS, TCM and adjacency lists, the result is shown in Table I. Because the update speed changes little with the matrix size, we fix the matrix size to be 1000×1000 and the fingerprint size to be 16-bit. The TCM is still implemented with 256 times memory and 4 sketches. In each data set we insert all the edges into the data structure, repeat this procedure 100 times and calculate the average speed. The unit we use is Million Insertions per Second (Mips). From the figure we can see that the speed of GSS is similar to TCM, because though more memory accesses are needed, GSS computes less hash functions. Both of them are much higher than the adjacency list. As the update speed of adjacency lists is not constant, this gap will enlarge in larger graphs.

TABLE I
UPDATE SPEED (MIPS)

Data Structure	email-EuAll	web-NotreDame	cit-HepPh
GSS	2.34329	2.2887	2.3862
TCM	2.2111	2.10417	2.05776
Adjacency Lists	0.364043	0.578596	0.724812

I. Summary of Experimental Studies

After extensive experiments of GSS and TCM on three data sets, we can make following conclusions:

1)GSS can perform better than the state-of-the-art by orders of magnitudes in supporting basic query primitives and compound queries.

2)The buffer size of GSS is very small. The memory and time cost of GSSare barely influenced by it.

3) GSShas high update speed which is beyond 2 million insertions per second.

VII. CONCLUSION

Graph stream summarization is a problem rising in many fields. However, as far as we know, there are no prior work that can support all kinds of queries with high accuracy. In this paper, we propose graph stream summarization data structure Graph Stream Sketch (GSS). It has $O(|E|)$ memory usage where $|E|$ is the number of edges in the graph stream, and $O(1)$ update speed. It supports almost 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] 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.

[3] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 3, pp. 270–313, 2003.

[4] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *SIGMOD*, pp. 1449–1463, ACM, 2016.

[5] 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.

[6] P. Zhao, C. C. Aggarwal, and M. Wang, "gsketch: on query estimation in graph streams," *PVLDB*, vol. 5, no. 3, pp. 193–204, 2011.

[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] M. Elkin, "Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners," in *ICALP*, pp. 716–727, Springer, 2007.

[10] V. Braverman, R. Ostrovsky, and D. Vilenchik, "How hard is counting triangles in the streaming model?," in *ICALP*, pp. 244–254, Springer, 2013.

[11] 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.

[12] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework.," in *OSDI*, vol. 14, pp. 599–613, 2014.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph-parallel computation on natural graphs.," in *OSDI*, vol. 12, p. 2, 2012.

[14] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, pp. 135–146, ACM, 2010.

[15] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012.

[16] S. Raghavan and H. Garcia-Molina, "Representing web graphs," in *ICDE*, pp. 405–416, IEEE, 2003.

[17] W. Fan, J. Li, X. Wang, and Y. Wu, "Query preserving graph compression," in *SIGMOD*, pp. 157–168, ACM, 2012.

[18] D. A. Spielman and N. Srivastava, "Graph sparsification by effective resistances," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1913–1926, 2011.

[19] J. Gao, C. Zhou, J. Zhou, and J. X. Yu, "Continuous pattern detection over billion-edge graph using distributed framework," in *ICDE*, pp. 556–567, IEEE, 2014.

[20] C. Wang and L. Chen, "Continuous subgraph pattern search over graph streams," in *ICDE*, pp. 393–404, IEEE, 2009.

[21] C. Song, T. Ge, C. Chen, and J. Wang, "Event pattern matching over graph streams," *PVLDB*, vol. 8, no. 4, pp. 413–424, 2014.

[22] P. LEcuyer, "Tables of linear congruential generators of different sizes and good lattice structure," *Mathematics of Computation*, vol. 68, no. 225, pp. 249–260, 1999.