HourglassSketch: An Efficient and Scalable Framework for Graph Stream Summarization

Jiarui Guo*, Boxuan Chen*, Kaicheng Yang*, Tong Yang*, Zirui Liu*, Qiuheng Yin*, Sha Wang†, Yuhan Wu*, Xiaolin Wang*, Bin Cui*, Tao Li†, Xi Peng‡, Renhai Chen‡, and Gong Zhang‡
*Peking University, China †National University of Defense Technology, China ‡Huawei Technologies Co., Ltd., China
{ntguojiarui, ykc, yangtong, zirui.liu, yuhan.wu, wxl, bin.cui}@pku.edu.cn,
{2100012923, yinqiuheng}@stu.pku.edu.cn, {ws0623zz, taoli_network}@163.com,
{pancy.pengxi, chenrenhai, nicholas.zhang}@huawei.com

***Abstract*—Graph stream is a special kind of data stream, where every item coming in sequence represents an edge in a dynamic graph. Graph stream has wide application in many fields, including cyber security, social networks and financial fraud detection. In this paper, we propose HourglassSketch, a two-stage data structure, for high-accuracy graph stream summarization. In Stage 1, HourglassSketch uses a CocoSketch to accurately record a partial collection of large-weight edges. In Stage 2, HourglassSketch integrates a TowerSketch with a TCMSketch to approximately record the statistics of most small-weight edges. In addition, we propose a key technique named Error Funnel to further reduce its error margin. Theoretical analysis and experimental results demonstrate that HourglassSketch supports various kinds of query operation and adapts well to graph stream storage. HourglassSketch achieves up to 100× smaller error and 2.7× higher speed than prior work. We also explore the versatility of HourglassSketch as a hardware-friendly framework by implementing it on FPGA and P4 platforms. We have released our codes on GitHub.**

I. Introduction

A. Background and Motivation

Graph is a structure composed of nodes and edges, and it is generally recognized for its succinct representation of pairwise relationships [1], [2]. In the era of big data, graph stands out for its unparalleled capacity to map and interpret the vast networks of interconnected data, which offers a clear and efficient way to visualize complex connections. *Graph stream*, which combines the feature of both graph and data stream to form a dynamic graph, is characterized by a series of items arriving in sequence. Each item in the graph stream represents an edge (u, v) in the graph, sometimes together with a weight w or a timestamp t [3]–[5]. Due to the universal representation of graph stream, it has widespread application across various of fields, including network monitoring [6], [7], social network analysis [8], [9], recommendation systems [10], [11] and traffic management [12]–[14].

However, graph storage is challenging in graph stream models: many real-world graphs are of huge size [15], [16], and we have to store the graph in a memory-efficient approach to avoid taking up too much space. In addition, we have to achieve high insertion throughput to catch up with high-speed items in the graph stream [17], [18]. Finally, in order to reflect as much information from the streaming graph as possible, we have to support various types of query operations.

Specially, we take the following two types of queries into consideration in graph stream scenarios in this paper:

- Edge query: Given an arbitrary edge (u, v), report its total weight $f(u, v)$;
- Node query: Given an arbitrary node u, report the sum weight $f(u)$ of all edges with tail (head) u.

B. Prior Art and Limitations

We divide existing algorithms on graph stream storage into two categories: data stream algorithms and graph stream algorithms.

Data stream algorithms are mainly designed based on data stream scenarios rather than graph stream scenarios, where every item in the data stream has a unique key. Since the edge (u, v) can also be viewed as a key, data stream algorithms can be used to store graph streams as well. According to their design goal, these algorithms can be further categorized as point query algorithms and subset query algorithms. The former aims to accurately estimate the frequency of any given key, and many sketches belong to this category [19]–[28]. The latter aims to estimate the aggregate frequency of an arbitrary subset by sampling, and these algorithms include Unbiased Space-Saving (USS) [29] and CocoSketch [30].

However, although these data stream algorithms are attractive in theory, none of them can be directly used for high-efficient graph stream summarization. As to point query algorithms, many sketches mainly focus on frequent items, and they achieve low error for these frequent items at the sacrifice of infrequent items, so they ignore edges with small weight when applied to graph stream. Also, many point query algorithms are not designed for subset query, and they have to traverse the estimated subset to answer subset query, so they

Jiarui Guo, Boxuan Chen, Kaicheng Yang, Tong Yang, Zirui Liu, Qiuheng Yin, Yuhan Wu, Xiaolin Wang and Bin Cui are with School of Computer Science, Peking University, Beijing, China.

Sha Wang and Tao Li are with College of Computer, National University of Defense Technology, Changsha, China.

Xi Peng, Renhai Chen and Gong Zhang are with Theory Lab, Central Research Institute, 2012 Labs, Huawei Technologies Co., Ltd., Hong Kong SAR, China.

The first three authors contribute equally. Corresponding author: Tong Yang (yangtong@pku.edu.cn).

cannot be used for subset query if the size of the subset is large, thus not suitable for node query in graph streams. As to subset query algorithms, they rely on a large subset to lower the variance of the estimation. Consequently, subset query algorithms fail to achieve high accuracy when subset query degenerates into point query with only one key in the subset, thus are not appropriate for edge query in graph streams.

Graph stream algorithms are generally designed based on graph stream scenarios. These algorithms include TCMSketch [31], Auxo [32], GSS [33], gMatrix [34], gSketch [35], MoSSo [36], Wind-Bell Index [37], LLAMA [38], Sortledton [39], TEGRA [40], CommonGraph [41], CuckooGraph [42] *etc.*. The idea of these algorithms is to compress the graph in their data structure within tight memory constraint and support fast query. However, the distribution of real graph stream is usually skew [43]–[46], but TCMSketch does not utilize this prior distribution to save memory. Also, some edges might appear in the graph stream more than once, but MoSSo and Wind-Bell Index cannot cope with parallel edges. In addition, many graph stream algorithms are designed for accurate graph storage, and they store the entire graph without approximation, which is memory-consuming. Even if these algorithms perform well within large memory, experimental results show that their performance drop sharply if allocated smaller memory, so they all have room for improvement in graph stream scenarios.

Recently, time windows has become a popular approach for managing large data streams by focusing on most recent data within a predefined time frame [47]–[51]. This method helps reduce memory constraint by discarding older data that is considered less relevant. Nonetheless, time windows technique does not necessarily work on graph stream scenarios, mainly because it merely records latest data, while many graph-related tasks, *e.g.* detecting community evolution [52] or maintaining the global connectivity of a network [53] rely on tracking of long-term graph features. Secondly, time windows lack the flexibility of capturing the continuous changes of the dynamic graph. They fail to reflect real-time shifts in connectivity or the emergence of key nodes, thus are less suitable for graph stream summarization.

C. Our Proposed Solution

In this paper, we propose HourglassSketch for storing graph stream. HourglassSketch maintains the information of the graph stream within small memory and simultaneously achieves high accuracy and throughput. Moreover, HourglassSketch is hardware-friendly: it can be implemented on FPGA and P4 platforms, which many graph storage solutions fail to achieve.

The design of HourglassSketch is based on two observations: **1)** To make it suitable for graph stream summarization, HourglassSketch has to support both edge query (*i.e.* query the weight of a given edge) and node query (*i.e.* query the aggregated weight of edges with a given tail/head). **2)** To cater for sparse graphs [54]–[56], HourglassSketch has to use sub-linear memory and separate large-weight and small-weight edges apart. In addition, HourglassSketch takes advantage of both data stream algorithms and graph stream algorithms for efficient graph storage.

HourglassSketch is composed of two stages: Stage 1 is a modified CocoSketch [30], which is designed for node query and mainly records edges with large weight; Stage 2 is a TowerTCMSketch, which originates from TowerSketch [22] but shares similar data structure with TCMSketch [31]. It is designed for edge query and mainly records edges with small weight. All items in the graph stream will be first inserted into Stage 1, and items which fail to enter Stage 1 will be inserted into Stage 2 instead. In this way, HourglassSketch supports edge query by getting its weight in both stages, and supports node query by finding all edges with given tail (head) node in Stage 1 and summing up their weight.

However, although the basic version of HourglassSketch fulfills the need of graph stream summarization, it still has room for improvement in terms of accuracy. Therefore, we propose an important optimization method to improve accuracy: We add an **Error Funnel** in HourglassSketch. Once an edge successfully enters Stage 1, its counter in the highest level of Stage 2 will be frozen if possible. Frozen counters cannot be incremented due to hash collision, and the increment operation will be carried on Error Funnel instead. This operation has two advantages: Firstly, partial information of large-weight edges are sometimes kept in Stage 2 due to insertion failure in Stage 1. We can separate large-weight edges from small-weight edges by recording the former in Stage 2, and the latter in Error Funnel; Secondly, TowerTCMSketch originates from TowerSketch, and both sketches suffer from overestimation error. By freezing the counter as early as possible, HourglassSketch prevents the accumulating of error as time goes by and achieves more accurate estimation.

In general, the data structure of HourglassSketch is like an hourglass: Error Funnel is implemented between Stage 1 and Stage 2, and all items not inserted into Stage 1 will be inserted into Stage 2 through Error Funnel. Besides, the work logic of HourglassSketch is unidirectional: just like hourglass, items can only enter Stage 2 from Stage 1, and cannot enter in reverse. This property ensures that HourglassSketch is hardware-friendly and can be deployed on FPGA and P4 tofinos (See our technical report [57] for more details).

This paper makes the following contributions:

- **(§III)** We propose HourglassSketch, a novel data structure for graph stream summarization. It is small, fast and accurate, and supports both edge query and node query over streaming graphs.
- **(§IV)** We theoretically prove the superiority of HourglassSketch over TCMSketch and show its error bound and time complexity.
- **(§V)** We conduct extensive experiments on different datasets. Experimental results show that HourglassSketch greatly outperforms existing algorithms in terms of accuracy and error.
- **(Appendix C, [57])** We further extend HourglassSketch to hardware platforms to verify its deployment flexibility on FPGA and P4 platforms.

II. Background and Related Work

A. Problem Definition

The symbols frequently used in this paper and their meanings are shown in Table I.

TABLE I: Symbols frequently used in this paper.

Notation	Meaning
S	The graph stream
(u, v)	An arbitrary edge in the graph stream
w	Weight of an edge
d	Number of arrays in Stage 1
m	Number of buckets in each array in Stage 1
$h_i(.,.)$	i^{th} hash function in Stage 1
A_i	i^{th} array in Stage 1
s	Number of arrays in Stage 2
n_j	Number of rows (columns) in j^{th} array in Stage 2
δ_j	Size of counters in j^{th} array in Stage 2
$r_j(.)$	j^{th} row hash function in Stage 2
$c_j(.)$	j^{th} column hash function in Stage 2
B_j	j^{th} array in Stage 2
k	Parameter of Error Funnel

We now formally provide the definition of data stream and graph stream below.

Definition 1. ***Data Stream.*** *A data stream S is a series of items $\{e_1, e_2, \cdots, e_n, \cdots\}$ appearing in sequence. In this paper, every item e_i is a key-value pair (k, v). We assume that items with the same key can appear multiple times in the data stream.*

Definition 2. ***Graph Stream.*** *A graph stream S is a series of items $= \{e_1, e_2, \cdots, e_n, \cdots\}$ appearing in sequence. In this paper, every item e_i is an edge (u, v) with weight w in a directed graph, where u denotes the tail of the edge, and v denotes the head of the edge. We assume that an edge can appear multiple times in the graph stream, and the weight of every item is always positive.*

B. Related Work

CMSketch and TowerSketch are traditional sketches that are used in frequency estimation (point query), and CocoSketch is regarded as the best sketch solution for subset query. Finally, TCMSketch is the first generalized sketch on graph query. Since these sketches play an important role in our algorithm, and some of them are components of our algorithm, we now introduce these algorithms in detail in this section.

1) CMSketch and TowerSketch:

CMSketch [19] is composed of d counter arrays. Each array has m counters and is associated with a hash function. When inserting an item e, CMSketch uses these d hash functions to locate d counters and increments these counters by 1. When querying its frequency, CMSketch again checks these d counters and returns the minimum value as its frequency.

TowerSketch [22] still has d counter arrays but uses different-sized counters in different arrays. The lower array has more counters which are smaller in size, and the higher array has fewer counters which are larger in size. When inserting an item e, TowerSketch locates d hashed counters and only increments non-overflowed counters. When querying its frequency, TowerSketch returns the minimum value of d hash counters which are not overflowed. Since TowerSketch still allocates the same size of memory for different array, it has more small counters for infrequent items, so TowerSketch estimates their frequency more accurately.

2) CocoSketch:

CocoSketch [30] is composed of d bucket arrays, and each bucket array has m buckets. Each bucket records the item and its frequency (weight). To insert an item e with weight w, CocoSketch uses d hash functions to locate one bucket in each array. If e is already recorded in any bucket, we just increment its frequency by w and return; Otherwise, we select the bucket with the minimum frequency among these d buckets (suppose the item in the bucket is e' and its frequency is $f_{e'}$). We increment its frequency by w. Then, with probability $\frac{w}{f_{e'}}$, e will replace e'. In this way, CocoSketch provides unbiased frequency estimation for both an arbitrary item and an arbitrary subset.

3) TCMSketch:

The idea of TCMSketch [31] originates from CMSketch. TCMSketch is composed of s counter arrays, but each array is a 2-dimensional array with n rows and n columns. Each row is associated with a hash function r_i, and each column is associated with a hash function c_i. To insert an edge (u, v) with weight w in the graph stream, we use r_i and u to locate a row, and c_i and v to locate a column. The counter at $(r_i(u), c_i(v))$ in each array will be incremented by w. When querying its frequency, TCMSketch again uses these hash functions to locate d counters, and returns the minimum value as its weight.

III. The HourglassSketch Algorithm

In this section, we introduce the algorithm of HourglassSketch. We first propose the basic version of HourglassSketch, which is composed of a CocoSketch in Stage 1 and a TCMSketch in Stage 2. Then we replace the TCMSketch with TowerTCMSketch in Stage 2 to achieve higher accuracy for small-weight edges. Finally, we introduce the idea of Error Funnel and propose the final version of HourglassSketch.

A. The Basic Version

The design idea of HourglassSketch is to separate large-weight edges from small-weight edges, and record these edges in two stages respectively. Specifically, HourglassSketch consists of two stages: Stage 1 and Stage 2. Stage 1 is a CocoSketch to record edges with large weight and supports node query; Stage 2 is a TCMSketch to record edges with small weight.

Data Structure: In the basic version of HourglassSketch, Stage 1 is a modified CocoSketch with d arrays. Each array is composed of m buckets and each array is associated with a hash function $h_i(.,.)$. Each bucket has three fields: key field, which records the key of an edge; coco counter, which is just the counter in traditional CocoSketch; and pure counter, which records the true weight of the edge after entering the bucket.

Stage 2 is a TCMSketch with s arrays. Each counter array has n rows and n columns and is associated with a row hash function $r_i(.)$ and a column hash function $c_i(.)$.

Insertion Operation: To insert an edge (u, v) with weight w into HourglassSketch, we first use (u, v) to query Stage 1. We use d hash functions $h_1(u, v), \cdots, h_d(u, v)$ to locate d buckets $A_1[h_1(u, v)], \cdots, A_d[h_d(u, v)]$. Similar to CocoSketch, there are two sub-cases:

Case 1: If (u, v) matches the key in any of the d buckets, we simply increment its coco counter and pure counter by w and return.

Case 2: If (u, v) does not match the key in any of the d buckets, we find the bucket with minimum coco counter among these d buckets (suppose it is $A_k[h_k(u, v)]$ and the key field in the bucket is (u', v')). We increase its coco counter by w. And then, with probability $\frac{w}{A_k[h_k(u,v)].coco}$, (u, v) is inserted into the bucket. If (u, v) successfully enters the bucket, we insert (u', v') and the value of pure counter into Stage 2 and reset the pure counter as w; Otherwise, we just insert (u, v) and its weight w into Stage 2.

Query Operation: HourglassSketch supports both edge query and node query.

1) To query an edge (u, v), we need to query both Stage 1 and Stage 2. Stage 1 returns the coco counter C and the pure counter P (if (u, v) is not recorded in Stage 1, then $C = P = 0$), and Stage 2 returns the minimum value of d mapped counters T. We report three kinds of estimation:

- an unbiased value $\hat{f}(u, v) = C$;
- an overestimation value $\bar{f}(u, v) = P + T$;
- an underestimation value $\underline{f}(u, v) = P$.

2) To query a node u, HourglassSketch only reports an unbiased value: we traverse Stage 1 to find all edges with tail u and sum up their coco counter, *i.e.*

$$\hat{f}(u) = \sum_{A_i[j].key=(u,.)} A_i[j].coco.$$

A Running Example: For simplicity, we assume $d = 2, m = 3, s = 2$ and $n_1 = n_2 = 3$. Figure 1 shows a running example of insertion procedure of HourglassSketch. To insert an edge (u_1, v_1) with weight 2, we first use a hash function to map (u_1, v_1) into one bucket in each array in Stage 1. Since (u_1, v_1) is already recorded in A_2, we just increment the coco counter and the pure counter of the bucket by its weight 2 and return. To insert an edge (u_2, v_2) with weight 10, we again map one bucket in each array in Stage 1. Since (u_2, v_2) does not match any edge recorded in the bucket, we find the bucket with minimum coco counter among the mapped buckets (here it is the second bucket in A_2). We first increment its coco counter by 10 (from 5 to 15). Then, with probability $\frac{10}{15}$, (u_2, v_2) will replace the edge (u_3, v_3) and enter the bucket by setting its pure counter as 10, while (u_3, v_3) with weight 2 will be inserted into Stage 2. With probability $\frac{5}{15}$, (u_2, v_2) fails to enter the bucket, and will be inserted into Stage 2 instead. Suppose (u_2, v_2) is successfully inserted into the bucket, then (u_3, v_3) will be inserted into Stage 2. We map (u_3, v_3) into one counter in each array, and the mapped counters will be incremented by its pure counter (here is 2).

B. From TCMSketch to TowerTCMSketch

The basic version of HourglassSketch can fulfill the need of graph stream summarization. However, Stage 2 is a TCMSketch, which originates from CMSketch and fails to achieve high accuracy in data stream scenarios. Since TowerSketch is the state-of-the-art algorithm on frequency estimation [58], we propose TowerTCMSketch based on TowerSketch to achieve high accuracy.

Data Structure: TowerTCMSketch combines the feature of both TowerSketch and TCMSketch simultaneously. It is consisted of s counter arrays. However, in TowerTCMSketch, different arrays use different size of counters, and have different number of rows (columns). Specifically, the i^{th} array is a 2-dimensional array B_i with n_i rows and n_i columns and is associated with a row hash function $r_i(.)$ and a column hash function $c_i(.)$. The size of counters in i^{th} array is δ_i bits.

Operation: To insert an edge (u, v) with weight w, we use d row hash functions and d column hash functions to locate d counters $B_1[r_1(u)][c_1(v)], \cdots, B_s[r_s(u)][c_s(v)]$. All these counters will be incremented by w, and if any of the counter will overflow after increment (*e.g.* the value in $B_j[r_j(u)][c_j(v)]$ exceeds $2^{\delta_j} - 1$), we just mark it as an overflowed counter by setting its value as $2^{\delta_j} - 1$. To query the aggregated weight of an edge (u, v), we again uses $2d$ hash functions to find these d counters. TowerTCMSketch will return the minimum value among d hashed counters which are not overflowed.

C. Optimization: Error Funnel

To query the total weight of an edge, HourglassSketch has to query both Stage 1 and Stage 2 to get an overestimation value. However, Stage 2 originates from TowerSketch (CMSketch) and suffers from overestimation error in practice. Also, we notice that an edge (u, v) will only be inserted into Stage 1 if it is recorded in Stage 1, so its associated counter in Stage 2 will only be incremented due to hash collision. Therefore, we propose the **Error Funnel**, inspired by OneSketch [23] and Double-Anonymous Sketch [24]. Its key idea is to freeze the counter associated with edge (u, v) in Stage 2 after it enters Stage 1, and unfreeze the counter if (u, v) is evicted from Stage 1. Frozen counter cannot be incremented due to hash collision, so HourglassSketch achieves higher accuracy for these edges.

Data Structure: The data structure of Error Funnel is just like a funnel. Let $t = n_s^2$ be the number of counters in B_s in Stage 2, and k be the parameter of Error Funnel. For simplicity, we assume that t is a power of 2. Error Funnel is divided into several levels. In the highest level (Level 1), every 2^k adjacent counters in B_s are grouped together, and every group has a freezing bucket. In Level 2, every 2^{k+1} adjacent counters in B_s are grouped together, and every group has a freezing bucket. Similarly, in Level l of Error Funnel, every 2^{k+l-1} adjacent counters in B_s are grouped together, and in

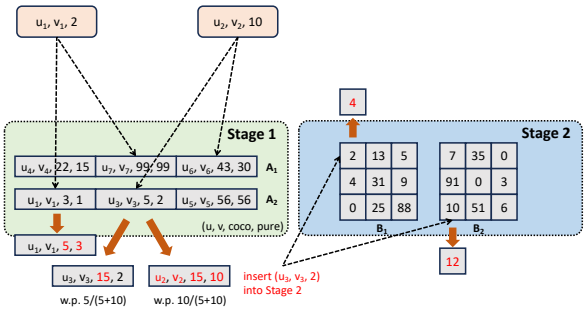


Fig. 1: Data Structure and Examples of HourglassSketch

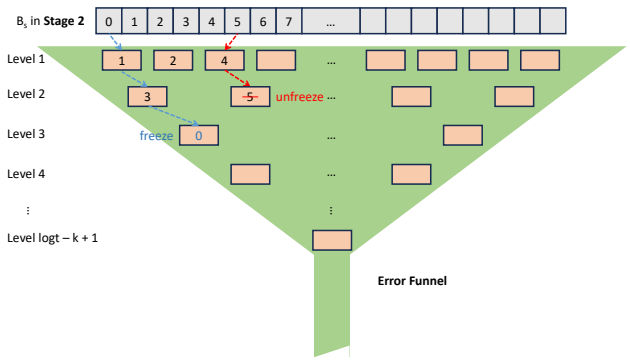


Fig. 2: Data Structure and Examples of Error Funnel

the lowest level (Level $\log t - k + 1$) there is only one group and one freezing bucket. Every freezing bucket has two fields: index field, which records the index of the frozen counter, and counter field, which records the weight of all collided edges after the counter is frozen.

Operation: The Error Funnel operation is triggered when a new edge (u, v) successfully replace an edge (u', v') in Stage 1. In this situation, HourglassSketch will try to freeze the counter for (u, v) and unfreeze the counter for (u', v').

1) To freeze the counter for (u, v), HourglassSketch checks all freezing buckets in the group that $B_s[r_s(u)][c_s(v)]$ belongs to in Error Funnel, starting from Level 1. If Error Funnel finds a freezing bucket which records $(r_s(u), c_s(v))$ in index field, then the counter has already been frozen before, so HourglassSketch just return; If Error Funnel finds an empty bucket, then the freezing bucket will be occupied by (u, v) by setting its index field as $(r_s(u), c_s(v))$ and counter field as 0. After $B_s[r_s(u)][c_s(v)]$ is frozen, any increment on $B_s[r_s(u)][c_s(v)]$ will be conducted on the counter field in the freezing bucket instead of $B_s[r_s(u)][c_s(v)]$.

2) To unfreeze the counter for (u', v'), HourglassSketch again checks all freezing buckets in the group that $B_s[r_s(u')][c_s(v')]$ belongs to in Error Funnel. Error Funnel will sum up the counter field of all buckets whose index field corresponds with edge (u', v'). These associated freezing buckets will be cleared, and the information of these buckets will be transferred to Stage 2: we find the maximum value of the sum result and the pure counter of (u', v'), and increment $B_s[r_s(u')][c_s(v')]$ by this maximum value.

A Running Example: For simplicity, we assume $k = 1$, so two counters are grouped together in Level 1 (the highest level) in Error Funnel. Figure 2 shows a running example of Error Funnel. Numbers in each grey box shows the index of the counter, and numbers in each orange box shows the index field of the freezing bucket. To freeze counter $B_s[0]$, Error Funnel checks all freezing buckets associated with $B_s[0]$, starting from Level 1. Error Funnel first finds an empty freezing bucket in Level 3, so Error Funnel freezes counter $B_s[0]$ in this freezing bucket by setting its index field as 0, and any further increment on $B_s[0]$ will be conducted on its counter field instead. To unfreeze counter $B_s[5]$, Error Funnel again checks all freezing buckets associated with $B_s[5]$. The index field of the freezing bucket in Level 2 is just 5, which shows that the counter was freezed in this freezing bucket before. Therefore, Error Funnel unfreezes $B_s[5]$ by obtaining its counter field and clearing this freezing bucket. Finally, Error Funnel gets the maximum value of the counter field and the pure counter of the edge, and adds it to $B_s[5]$ in Stage 2.

D. Our Final Version

Our final version of HourglassSketch is consists of two stages as above. Stage 1 is a CocoSketch, and Stage 2 is a TowerTCMSketch. In addition, the Error Funnel is added to improve accuracy. The full operation of HourglassSketch can be summarized as follows:

Insertion Operation: To insert an edge (u, v) with weight w, HourglassSketch first checks Stage 1.

Case 1: (u, v) is already recorded in a bucket. In this situation, we just insert (u, v) into Stage 1 and return.

Case 2: (u, v) collides with an edge (u', v'), and (u, v) replaces (u', v') in Stage 1. In this situation, we freeze the counter for (u, v) and unfreeze the counter for (u', v') in Error Funnel, which is shown in Section III-C.

Case 3: (u, v) collides with an edge (u', v'), and (u, v) does not replace (u', v') in Stage 1. In this situation, we just insert (u, v) into Stage 2. We first insert (u, v) into all arrays of Stage 2 except the last array (*i.e.* B_s). Then, we check the Error Funnel: if the index of the hashed counter in B_s (*i.e.* $B_s[r_s(u)][c_s(v)]$) is already frozen, we insert (u, v) into the freezing counter in the freezing bucket; otherwise, we insert (u, v) into B_s by increment $B_s[r_s(u)][c_s(v)]$ by w.

Query Operation: The final version of HourglassSketch also supports both edge query and node query.

1) To query an edge (u, v), HourglassSketch first query Stage 1. Suppose Stage 1 returns the coco counter C and the pure counter P. We then query Stage 2 and Error Funnel. Suppose Stage 2 returns the minimum value of d mapped counters T, and the sum of counter field in freezing buckets with index field equal to $(r_s(u), c_s(v))$ in Error Funnel is F. We report four kinds of estimation:

- an unbiased value $\hat{f}(u, v) = C$;
- an overestimation value $\bar{f}(u, v) = P + T + F$
- a likely overestimation value $\tilde{f}(u, v) = P + T$;
- an underestimation value $\underline{f}(u, v) = P$.

Note that if (u, v) is not recorded in Stage 1, then only the overestimation value $\bar{f}(u, v)$ is meaningful, since $C = P = 0$ in this situation, and the information of (u, v) is mostly recorded in the funnel if its associated counter in B_s is frozen. **2)** To query a node u, HourglassSketch only reports an unbiased value: we again traverse Stage 1, find all edges with tail u, and sum up their coco counter, just like the basic version.

E. Discussion: Innovative Techniques of HourglassSketch

Our proposed solution, HourglassSketch is composed of three components: CocoSketch in Stage 1, TowerTCMSketch in Stage 2, and Error Funnel in the intermediary layer. While the idea of these data structures are inspired by prior work, our approach combines them in novel ways that significantly improves efficiency and scalability in large-scale graph streams. To better showcase the novelty of HourglassSketch, we highlight the specific techniques introduced in each component as below:

Technique 1: Enhancing CocoSketch with the pure counter in Stage 1. CocoSketch is widely used in network measurement for accurate and unbiased subset query. However, two-stage sketches typically require precise weight tracking for heavy edges in Stage 1 and more approximate handling of light edges in Stage 2 [17], [21], [59], [60]. The unbiasedness of CocoSketch is designed for subset queries, which does not distinguish between heavy and light edges in the same way that a two-stage approach does. To cater for high-performance graph stream summarization within two-stage sketches, we modify CocoSketch in Stage 1 by adding pure counter in each bucket. Different from traditional CocoSketch with only one coco counter in each bucket, each bucket in Stage 1 now has two counters: coco counter just as traditional CocoSketch, and pure counter to maintain the true weight of an edge after it enters Stage 1. In this way, HourglassSketch achieves accurate edge query by accessing the pure counter, and still supports node query by accessing the coco counter.

Technique 2: Combining TowerSketch and TCMSketch in Stage 2. TowerSketch is widely used in network systems for efficiently managing flow size through its layered structure, and TCMSketch is the first sketch specially designed for handling large-scale graph streams. However, although both sketches are considered state-of-the-art in their respective fields, no prior work has explored the possibility of combining them together to achieve more precise graph stream summarization. We introduce TowerTCMSketch that integrates the strengths of both sketches: TowerTCMSketch inherits the feature of TCMSketch of hashing u and v separately to efficiently store graph streams; In addition, TowerTCMSketch utilizes the idea of different-size counters in different arrays to maintain high accuracy for light edges. The combination of both sketches allows TowerTCMSketch to record edges with low error, thus improving query accuracy for both heavy and light edges in graph streams.

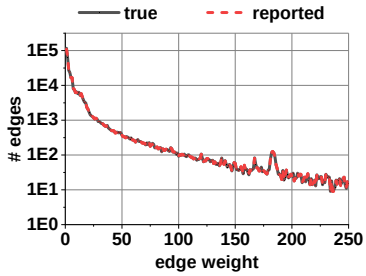


Fig. 3: Comparison of True and Reported Distribution

Technique 3: Implementing hardware-friendly Error Funnel for error reduction. Two-stage sketches typically retrieve results from both parts and sum them up to return the query result. However, these sketches sometimes suffer from overestimation error caused by hash collision, making it crucial to design novel techniques to minimize such error. Prior work, such as OneSketch [23] and Double-Anonymous Sketch [24] introduced complex data structures and sophisticated operations to address hash collisions. While these algorithms achieve high accuracy, their logic is no longer unidirectional, resulting in reduced hardware compatibility and posing challenges for efficient hardware implementation. To address these challenges, we propose Error Funnel as a hardware-friendly approach to reduce the error in HourglassSketch: Error Funnel automatically separates heavy and light edges apart by recording partial information of heavy edges in Stage 2, and storing light edges in Error Funnel. Notably, HourglassSketch still maintains a unidirectional workflow: items pass through Stage 1, then Error Funnel, and finally Stage 2. This property guarantees that HourglassSketch reduces overestimation error without compromising hardware-friendliness.

In general, HourglassSketch integrates the advantages of different data structures in a novel approach to enhance precision and efficiency. We analyze the edge weight distribution of the real graph stream and the distribution after processed by HourglassSketch in Figure 3 using CAIDA dataset [61], and find that the two distributions are highly similar, demonstrating the potential of HourglassSketch as an efficient and scalable framework for accurate graph stream summarization.

IV. Mathematical Analysis

In this section, we propose the mathematical property of HourglassSketch. We first derive error bound for the basic version of HourglassSketch to show its superiority over TCMSketch. Then we analyze the TowerTCMSketch and show its error bound. Next, we analyze the optimized HourglassSketch with Error Funnel. Finally, we show the time complexity of HourglassSketch. Due to space constraint, we put detailed proof in our technical report on GitHub [57].

A. Analysis of the Basic Version

In this section, we show the error bound of TCMSketch and the basic version of HourglassSketch. Since the basic version of HourglassSketch uses a TCMSketch in Stage 2, we first cite the error bound for TCMSketch, which is proved in [31].

Theorem 3. *Let n be the number of rows (columns) in each array, and s be the number of arrays in TCMSketch. For an arbitrary edge (u, v), suppose no other edge has tail u or head v. Let $f(u, v)$ be its aggregated weight, and $\bar{f}(u, v)$ be its weight reported by TCMSketch. Let N_d be the sum of weight of all edges in the graph stream. For any small positive number ε, TCMSketch guarantees*

$$P(\bar{f}(u,v) \geq f(u,v) + \varepsilon) \leq \left(\frac{N_d}{\varepsilon n^2}\right)^s. \tag{1}$$

However, TCMSketch does not discuss the error bound if more than one edges share the same head (tail) in [31]. In fact, the following theorem holds in this situation:

Theorem 4. *For an arbitrary edge (u, v), let $f(u, v)$ be its aggregated weight, and $\bar{f}(u, v)$ be its weight reported by TCMSketch. Let N_s be the sum of weight of edges with head u or tail v, and N_d be the sum of weight of edges otherwise. For any small positive number ε, TCMSketch guarantees*

$$P(\bar{f}(u,v) \geq f(u,v) + \varepsilon) \leq \left(\frac{N_s}{\varepsilon n} + \frac{N_d}{\varepsilon n^2}\right)^s. \tag{2}$$

Remark. *If $N_s = 0$, then the RHS of Equation 2 is just $\left(\frac{N_d}{\varepsilon n^2}\right)^s$, so Theorem 3 is a special case of Theorem 4. Also, if both u and v are not heavy nodes (nodes with large weight), then N_s will be close to 0, so the RHS of Equation 2 is close to $\left(\frac{N_d}{\varepsilon n^2}\right)^s$. However, if u or v is a heavy node, then N_s will be very large, so TCMSketch does not perform well in this situation.*

Next, we prove that $\bar{f}(u, v) = P + T$ is indeed an overestimation value, and $\underline{f}(u, v) = P$ is indeed an underestimation value for the total weight of edge (u, v).

Theorem 5. *For an arbitrary edge (u, v), let $f(u, v)$ be its aggregated weight. We have*

$$\underline{f}(u,v) \leq f(u,v) \leq \bar{f}(u,v). \tag{3}$$

Corollary 6. *For an arbitrary edge (u, v), let $f(u, v)$ be its aggregated weight, and $\bar{f}(u, v)$ be the overestimation value reported by the basic version of HourglassSketch. Let N'_s be the sum of weight of edges inserted into Stage 2 with head u or tail v , and N'_d be the sum of weight of edges inserted into Stage 2 otherwise. For any small positive number ε, HourglassSketch guarantees*

$$P(\bar{f}(u,v) \geq f(u,v) + \varepsilon) \leq \left(\frac{N'_s}{\varepsilon n} + \frac{N'_d}{\varepsilon n^2}\right)^s. \tag{4}$$

Remark. *Since N'_s is usually much smaller than N_s, and N'_d is usually much smaller than N_d, Theorem 5 clearly shows the superiority of HourglassSketch over TCMSketch.*

B. Analysis of TowerTCMSketch

In this section, we first analyze the error bound of TowerTCMSketch, then we use its error bound to show the error bound of HourglassSketch after replacing TCMSketch with TowerTCMSketch in Stage 2.

Theorem 7. *Let n_j be the number of rows (columns) in j^{th} array in TowerTCMSketch, and s be the number of arrays in TowerTCMSketch. Suppose j^{th} array in TowerTCMSketch uses δ_j bits counters, and $\delta_1 \leq \delta_2 \leq \cdots \leq \delta_s$. For an arbitrary edge (u, v), let $f(u, v)$ be its aggregated weight, and $\bar{f}(u, v)$ be its weight reported by TowerTCMSketch. N_s and N_d are defined similarly as above. For any small positive number ε, when $2^{\delta_{t-1}} - 1 \leq f(u, v) + \varepsilon < 2^{\delta_t} - 1$, TowerTCMSketch guarantees*

$$P(\bar{f}(u,v) \geq f(u,v) + \varepsilon) \leq \prod_{j=t}^{s} \left(\frac{N_s}{\varepsilon n_j} + \frac{N_d}{\varepsilon n_j^2}\right). \tag{5}$$

Corollary 8. *For an arbitrary edge (u, v), let $f(u, v)$ be its aggregated weight, and $\bar{f}(u, v) = P+T$ be the overestimation value reported by HourglassSketch with TowerTCMSketch in Stage 2. $n_j, \delta_j, s, N'_s, N'_d$ are defined similarly as above. For any small positive number ε, when $2^{\delta_{t-1}}-1 \leq T+\varepsilon < 2^{\delta_t}-1$, HourglassSketch guarantees*

$$P(\bar{f}(u,v) \geq f(u,v) + \varepsilon) \leq \prod_{j=t}^{s} \left(\frac{N'_s}{\varepsilon n_j} + \frac{N'_d}{\varepsilon n_j^2}\right). \tag{6}$$

Remark. *Since Stage 1 filters large-weight edges in advance, edges inserted into Stage 2 usually has smaller weight. These edges has smaller t, and n_j is larger after we replace TCMSketch with TowerTCMSketch, so the RHS of Equation 6 is smaller than RHS of Equation 4, and the accuracy of HourglassSketch improves significantly (See in Section V for more details).*

C. Analysis of the Optimized Version

In this section, we first show that $\underline{f}(u, v)$ and $\bar{f}(u, v)$ are still underestimation and overestimation value of $f(u, v)$. Then we analyze the underestimation error of $\hat{f}(u, v)$. Finally we analyze the space cost for Error Funnel.

Theorem 9. *For an arbitrary edge, let $f(u, v)$ be its aggregated weight. We still have*

$$\underline{f}(u,v) \leq f(u,v) \leq \bar{f}(u,v). \tag{7}$$

Theorem 10. *Suppose (u, v) enters Stage 1 for the first time and is never evicted from Stage 1. Among all edges inserted into Stage 2* ***before*** *(u, v)* ***enters Stage 1****, let $\tilde{N}'_s$ be the sum of weight of edges with head u or tail v , and $\tilde{N}'_d$ be the sum of weight of edges otherwise. N'_s, N'_d and t are defined similarly in Corollary 6 and 8. For any small positive number ε, if (u, v) occupies an freezing bucket in Error Funnel, then HourglassSketch guarantees*

$$P(\tilde{f}(u,v) \geq f(u,v)+\varepsilon) \leq \left(\frac{\tilde{N}'_s}{\varepsilon n} + \frac{\tilde{N}'_d}{\varepsilon n^2}\right) \cdot \left(\frac{N'_s}{\varepsilon n} + \frac{N'_d}{\varepsilon n^2}\right)^{s-1}. \tag{8}$$

if we use TCMSketch in Stage 2, and

$$P(\tilde{f}(u,v) \geq f(u,v)+\varepsilon) \leq \left(\frac{\tilde{N}'_s}{\varepsilon n_s} + \frac{\tilde{N}'_d}{\varepsilon n_s^2}\right) \cdot \prod_{j=t}^{s-1} \left(\frac{N'_s}{\varepsilon n_j} + \frac{N'_d}{\varepsilon n_j^2}\right) \tag{9}$$

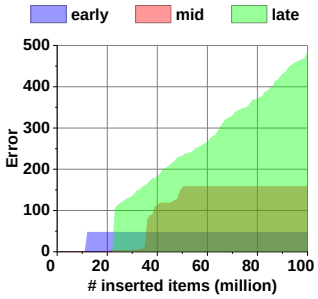


Fig. 4: Relationship between Error and Time

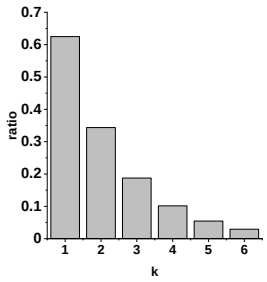


Fig. 5: Ratio of Memory Usage of Error Funnel to B_s

if we use TowerTCMSketch in Stage 2.

Based on Theorem 4 and Theorem 7, the error of TCMSketch and TowerTCMSketch arises from all edges colliding with (u, v), and Corollary 6 and Corollary 8 show that the error of HourglassSketch (basic version) arises from all colliding edges entering Stage 2. However, according to Theorem 10, the error of HourglassSketch (optimized version) only arises from colliding edges that enters Stage 2 before (u, v) enters Stage 1, and error will not increase after (u, v) occupies a bucket in Error Funnel. Hence, the optimization version of HourglassSketch has smaller error for heavy edges. In addition, the earlier an edge (u, v) enters Stage 1 and occupies a freezing bucket, the more accurate its weight estimation is.

To measure the property of Error Funnel, we generate three identical graph stream datasets with a fixed edge (u, v) coming at different time using CAIDA dataset [61]. We insert three datasets into HourglassSketch respectively and measure the error (*i.e.* $|\hat{f}(u, v)-f(u, v)|$) simultaneously. The experimental results (Figure 4) show that the error of (u, v) is the smallest if it comes very early. In this situation, few edges collides with (u, v), so $\hat{f}(u, v)$ is close to $f(u, v)$ and the error is constant after (u, v) enters Stage 1; the measurement error is larger if (u, v) arrives at the midpoint of the dataset, and the error peaks if (u, v) comes at a very late time. The Error Funnel is already full if (u, v) comes very late, so (u, v) cannot get a freezing bucket in Error Funnel, and its error accumulates to a very high level.

Theorem 11. *Assume that $t = n_s^2$ is a power of 2, and the size of the counter field in Error Funnel is δ'_s bits. Then the memory cost of Error Funnel is at most $(k+\delta'_s+1)\cdot 2^{\log t-k+1}$ bits.*

Remark. *Since Error Funnel usually record edges with small weight, δ'_s is usually smaller than δ_s. For example, when $\delta_s = 16, \delta'_s = 8$ and $k \geq 3$, the extra memory usage is smaller than 20% (see in Figure 5), so Error Funnel with small memory is capable of reducing the error of HourglassSketch.*

D. Time Complexity

Theorem 12. *Assume that we use TowerTCMSketch in Stage 2, and Error Funnel is used to reduced error. The time complexity to insert an edge (u, v) is at most $O(d+\log n_s-k+s)$.*

V. Experimental Results

In this section, we evaluate the performance of HourglassSketch through extensive experiments. First, we describe the experimental setup in Section V-A. Then, we compare the performance of HourglassSketch with state-of-the-art solutions and show its superiority in Section V-B. We show how parameter settings affect HourglassSketch performance in Section V-C. We also provide analyses on HourglassSketch in Section V-D. Finally, we implement HourglassSketch on Neo4j graph database as a case study to demonstrate its deployment flexibility in Section V-E. All related codes are released on GitHub anonymously [57].

A. Experimental Setup

Implementation: We implement HourglassSketch and all other algorithms in C++. In all experiments, we use Bob Hash [62] with different hash seeds to implement the hash functions.

Computation Platform: We conducted all the experiments on a server with one 18-core processor and 128 GB DRAM memory. The processor has 64KB L1 cache, 1MB L2 cache for each core, and 24.75MB L3 cache shared by all cores.

Datasets: We use four kinds of datasets. In all experiments, we regard the weight of all edges as 1 in the graph stream.

1) CAIDA Dataset: The CAIDA dataset is streams of anonymous IP traces collected from 2016 by CAIDA [61]. CAIDA identifies each flow by the five-tuples: source and destination IP address, source and destination port, protocol. We use source IP address as u and destination IP address as v. We use 100 million items.

2) DBLP Dataset: We extract 919763 authors as nodes and 10 million author-pairs as edges from DBLP archive [63]. Since the co-author relationship is mutual, we treat both (u, v) and (v, u) as edges in the directed graph if author u co-authors with author v.

3) Network Dataset: The network dataset [64] contains users' email communication history. Each item has three values u, v, t, which means user u sends an email to user v at time t. We use 420045 items.

4) Synthetic Dataset: We generate the Synthetic dataset following the Zipfian distribution with $\alpha = 1$. We use 100 million items.

Metrics: HourglassSketch supports both edge query and node query, so we measure the performance of HourglassSketch in both tasks. For simplicity, we only introduce the metrics used in edge query, and metrics used in node query can be defined similarly.

1) ARE: Let $f(u_1, v_1), \cdots, f(u_{|E|}, v_{|E|})$ be the real weight of all edges, and $\hat{f}(u_1, v_1), \cdots, \hat{f}(u_{|E|}, v_{|E|})$ be their estimated weight mentioned in Section III-D. The Average Relative Error (ARE) is defined as $\frac{1}{|E|}\sum_{i=1}^{|E|}\frac{|\hat{f}(u_i,v_i)-f(u_i,v_i)|}{f(u_i,v_i)}$.

2) AAE: Let $f(u_1, v_1), \cdots, f(u_{|E|}, v_{|E|})$ be the real weight of all edges, and $\hat{f}(u_1, v_1), \cdots, \hat{f}(u_{|E|}, v_{|E|})$ be their estimated weight. The Average Absolute Error (AAE) is defined as $\frac{1}{|E|}\sum_{i=1}^{|E|}|\hat{f}(u_i, v_i) - f(u_i, v_i)|$.

3) F1 Score: F1 Score is only used in top-k detection. It is defined as $\frac{2\times RR\times PR}{RR+PR}$, where $PR = \frac{\text{Reported top-}k}{\text{Reported edges}}$, $RR = \frac{\text{Reported top-}k}{k}$.

4) Throughput: We use million of operations (insertions and queries) per second (Mops) to measure the throughput. We repeat the experiment for 10 times and calculate the average results as our throughput.

B. Comparison with Prior Work

In this section, we run HourglassSketch on four datasets and compare HourglassSketch with three state-of-the-art solutions: TCMSketch [31], Auxo [32] and GSS [33]. We compare the performance of these algorithms on both edge query and node query. We also measure the running speed of these algorithms. As to top-k edge/node detection, we set k to 1% of the number of edges/nodes in the graph stream.

1) Comparison on Edge Query:

We conduct experiments on both per-edge weight estimation and top-k edge detection. The experimental results (Figure 6-8) show that HourglassSketch outperforms all state-of-the-art algorithms on edge query. On Network dataset, the AAE of HourglassSketch is only 0.83 within in 100KB memory, which shows that HourglassSketch performs well even within tight memory constraint. The results also demonstrate that HourglassSketch stands out on top-k edge detection. The F1 Score of HourglassSketch is greater than 0.8 when memory is 20MB on CAIDA dataset. On synthetic dataset, HourglassSketch finds almost all top-k edges when memory is greater than 40MB, and the ARE of weight estimation for top-k edges is smaller than 0.05.

2) Comparison on Node Query:

We conduct experiments on both per-node weight estimation and top-k node detection. The experimental results (Figure 9-11) show that HourglassSketch exhibits superior performance on node query. The AAE of HourglassSketch is on average 1133×, 46.7×, 3.7×, 43.67× smaller on four datasets on per-node weight estimation. On top-k node detection task, the F1 Score of HourglassSketch reaches 85%, and the ARE is close to 0.1 on CAIDA dataset within 40MB memory, while other algorithms do not find most top-k nodes.

3) Comparison on Throughput:

The experimental results (Figure 12-14) show that the throughput of HourglassSketch is only lower than TCMSketch. The insertion throughput of HourglassSketch is all higher than 2 Mops on four datasets. In addition, the edge query throughput of HourglassSketch is all higher than 3 Mops, and its node query throughput is the highest among the four algorithms. Even if TCMSketch queries the weight of an edge faster, it has to check the whole row (column) in each array for node query, hence its node query throughput falls off.

C. Experiments on Parameter Settings

In this section, we measure the effects of some key parameters of HourglassSketch, namely, the number of arrays in Stage 1 and the ratio of the memory usage of Stage 1 to the total memory usage $ratio$. We set the memory usage to 1800KB, 2400KB and 3000KB. We use $k = 2$, $s = 4$ and set $\delta_1 = 2, \delta_2 = 4, \delta_3 = 8, \delta_4 = 16$. We conduct experiments on CAIDA dataset, and use AAE of edges to evaluate the effects of these parameters.

Effects of d (Figure 15): *The experimental results show that the best option of d is 2.* In this experiment. we vary d from 1 to 6, and results show that the performance of HourglassSketch does not differ significantly when $d \geq 2$. However, since HourglassSketch has to check every array in Stage 1 to insert an item, HourglassSketch will have lower throughput if we set a larger d. As a result, we set $d = 2$ in other experiments.

Effects of $ratio$ (Figure 16): *The experimental results show that the best option of $ratio$ is among 0.1 and 0.2.* In this experiment, we vary $ratio$ from 0.05 to 0.3 in a step of 0.05. The results show that the performance of HourglassSketch peaks when $ratio$ is among 0.1 and 0.2. In fact, setting a larger $ratio$ can store large-weight edges more accurately in Stage 1. However, only a small portion of edges in the graph stream have large weight, so we need to allocate sufficient memory to Stage 2 to ensure that small-weight edges can be stored accurately. Taking both cases into account, we set $ratio = 0.1$ by default.

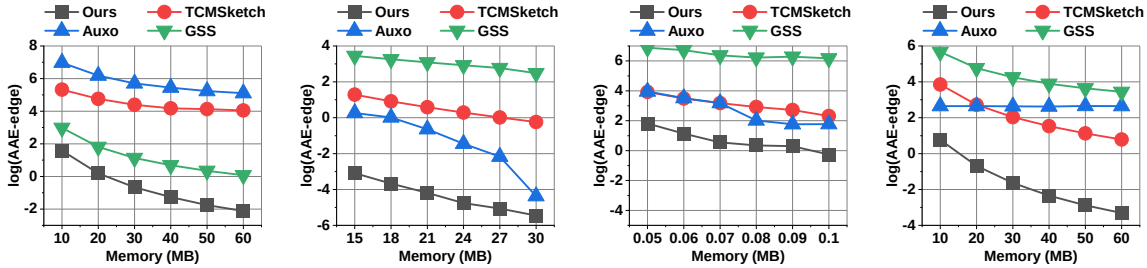
D. Analysis on HourglassSketch

In this section, we analyze the effect of TowerTCMSketch and Error Funnel in HourglassSketch. We also measure the error of HourglassSketch on light edges and nodes (*i.e.* edges and nodes with small weight) and on large graph streams. We conduct experiments on CAIDA dataset. Due to space constraint, we put experiments on large graph streams in our technical report on GitHub [57].

Effects of TowerTCMSketch (Figure 17): We conduct experiments with TCMSketch in Stage 2 and TowerTCMSketch in Stage 2 on per-edge weight estimation, and experimental results show that replacing TCMSketch with TowerTCMSketch will significantly lower ARE of edges. In fact, the superiority of TowerTCMSketch over TCMSketch can be explained by the truth that the overall accuracy of TowerSketch is much higher than CMSketch. Therefore, using TowerTCMSketch instead of TCMSketch will definitely reduce the error of HourglassSketch.

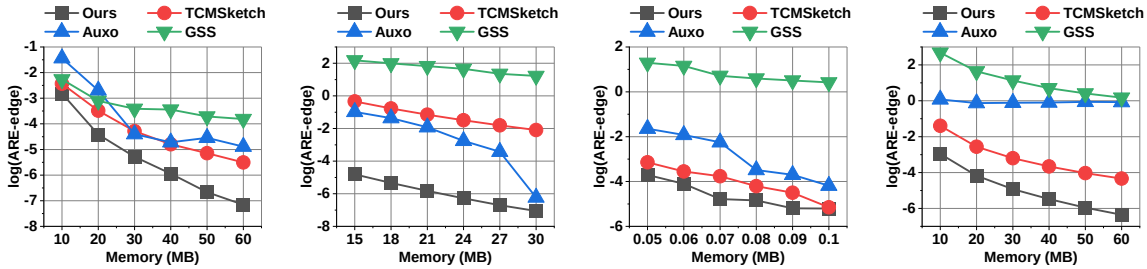
Effects of Error Funnel (Figure 18): We conduct experiments with and without Error Funnel in HourglassSketch on top-k edge detection, and experimental results show that adding Error Funnel can improve accuracy by 30% within tight memory. Even if TowerTCMSketch estimates the weight of edges accurately, it still has room for improvement in terms of large-weight edges. As TowerTCMSketch has larger error for these edges, Error Funnel can automatically separate edges with different weight apart. Error Funnel also slightly reduces top-k edge query error when memory is large.

Experiments on Light Edges and Nodes (Figure 19-20): We conduct experiments on light edges and nodes query on HourglassSketch and TCMSketch, and experimental results show that HourglassSketch achieves accurate estimation for



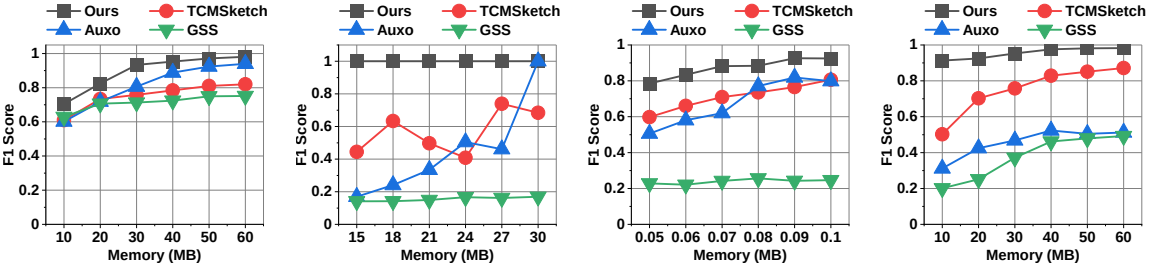
(a) CAIDA (b) DBLP (c) Network (d) Synthetic

Fig. 6: AAE of Edges on Different Datasets



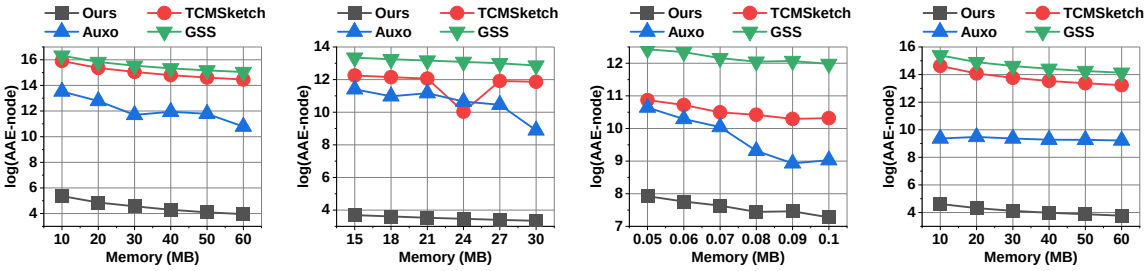
(a) CAIDA (b) DBLP (c) Network (d) Synthetic

Fig. 7: ARE of Top-k Edges on Different Datasets



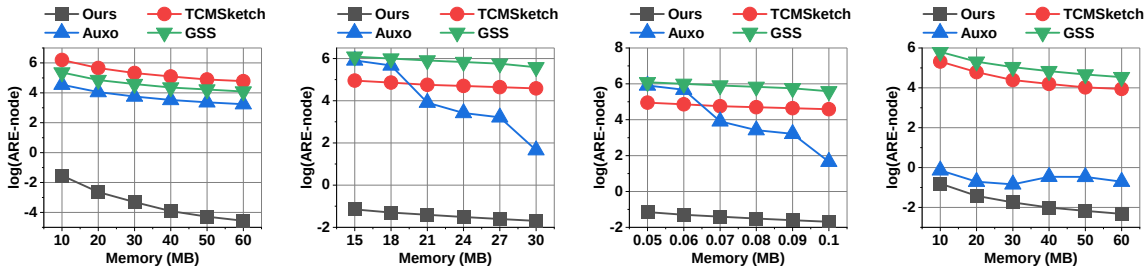
(a) CAIDA (b) DBLP (c) Network (d) Synthetic

Fig. 8: F1 Score of Finding Top-k Edges on Different Datasets



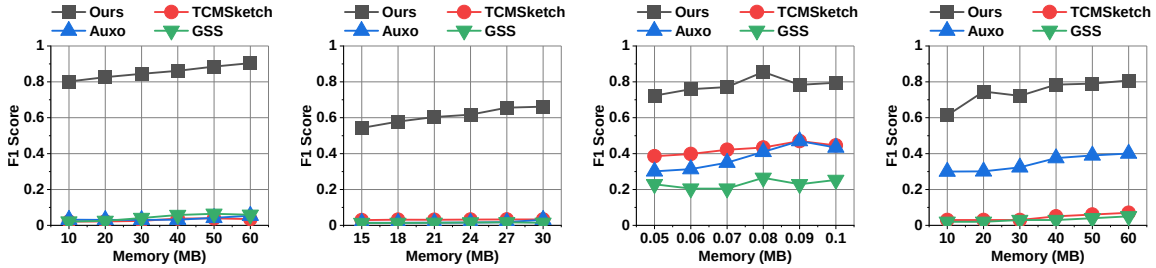
(a) CAIDA (b) DBLP (c) Network (d) Synthetic

Fig. 9: AAE of Nodes on Different Datasets



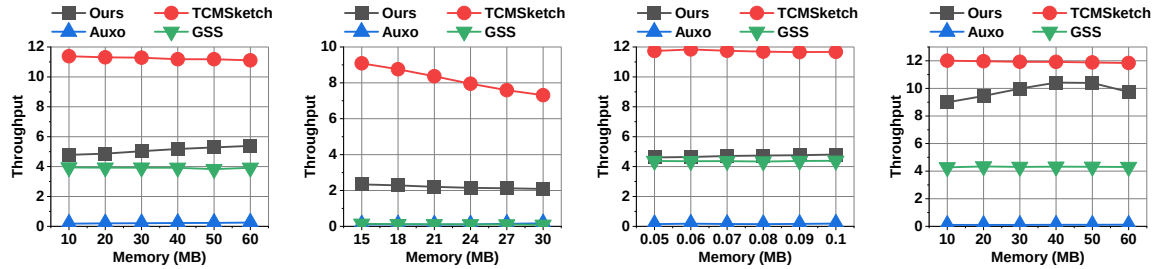
(a) CAIDA (b) DBLP (c) Network (d) Synthetic

Fig. 10: ARE of Top-k Nodes on Different Datasets



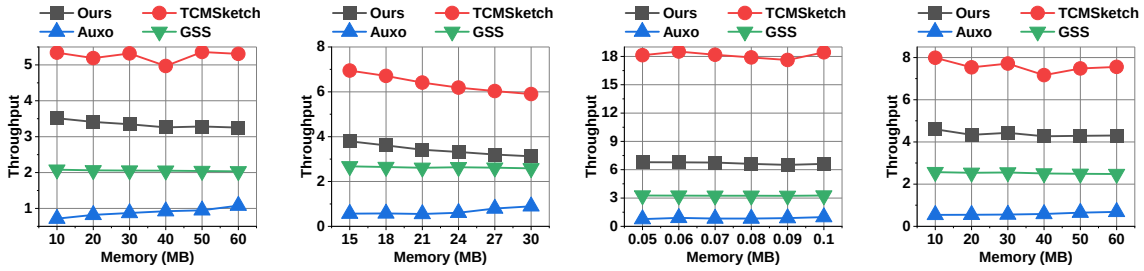
(a) CAIDA (b) DBLP (c) Network (d) Synthetic

Fig. 11: F1 Score of Finding Top-k Nodes on Different Datasets



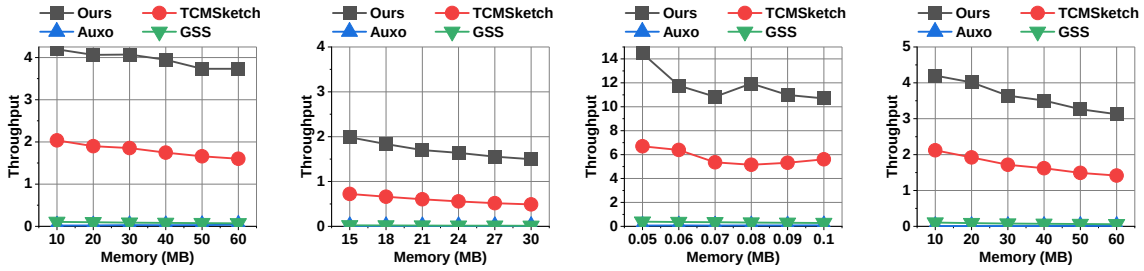
(a) CAIDA (b) DBLP (c) Network (d) Synthetic

Fig. 12: Insertion Throughput on Different Datasets



(a) CAIDA (b) DBLP (c) Network (d) Synthetic

Fig. 13: Edge Query Throughput on Different Datasets



(a) CAIDA (b) DBLP (c) Network (d) Synthetic

Fig. 14: Node Query Throughput on Different Datasets

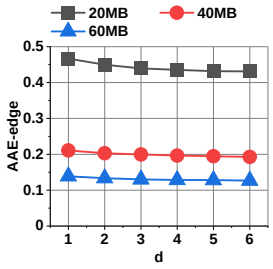


Fig. 15: Effects of d

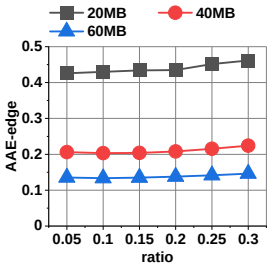


Fig. 16: Effects of $ratio$

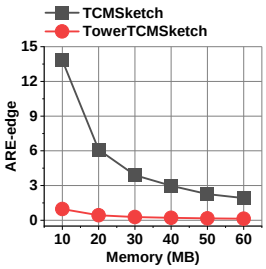


Fig. 17: Effects of TowerTCMSketch

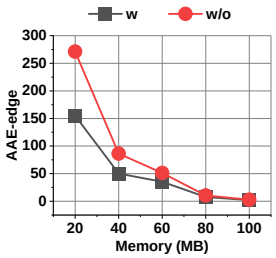
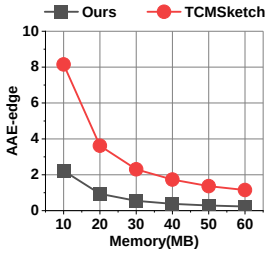
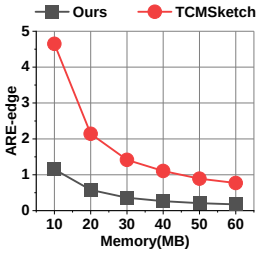


Fig. 18: Effects of Error Funnel

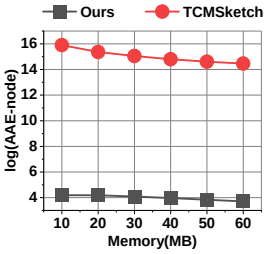


(a) AAE

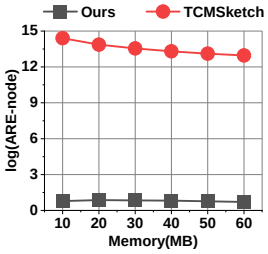


(b) ARE

Fig. 19: Experiments on Light Edges

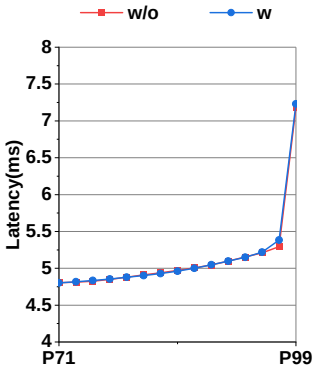


(a) AAE

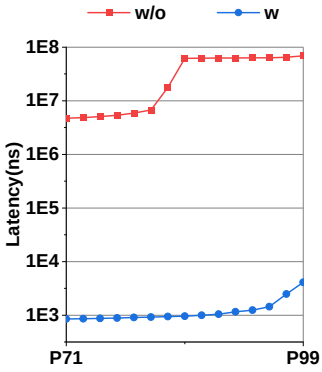


(b) ARE

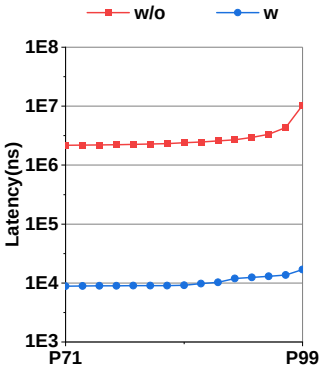
Fig. 20: Experiments on Light Nodes



(a) Insert



(b) Edge Query



(c) Node Query

Fig. 21: Tail Latency on Neo4j Graph Database

light edges and nodes, while TCMSketch fails to. The AAE of HourglassSketch on light edge query is smaller than 1 when memory is 20MB, as HourglassSketch accurately records light edges in Stage 2 using TowerTCMSketch. Also, TCMSketch has to sum up all counters in a row (column) for node query, and these counters tend to overflow due to hash collision, so its estimating error exceeds that of HourglassSketch.

E. Experiments on Neo4j Graph Database

In this section, we implement HourglassSketch to accelerate edge and node query in real graph databases.

Background: The Neo4j graph database [65] is a widely-used system known for its high performance, robustness and flexibility. To support various query operations, Neo4j stores the graph in the form of an adjacency list, which is well-suited for sparse graphs. However, answering edge and node queries requires traversing the entire list, which has room for improvement with HourglassSketch.

Implementation: Similar to prior work [37], we integrate an extra instance of HourglassSketch into Neo4j using Java. When inserting an edge (u, v) into Neo4j, we also insert it into HourglassSketch. To perform edge and node query, we query HourglassSketch to retrieve the results. We allocate 10MB memory for HourglassSketch, and use CAIDA dataset.

Experiments (Figure 21): We measure the tail latency of insertion, edge query and node query operation with and without HourglassSketch, and experimental results show that HourglassSketch significantly accelerates query operation in Neo4j graph database. The insertion latency of Neo4j remains almost unchanged after implementing HourglassSketch, with a worst-case slowdown of merely 0.1ms. In comparison, HourglassSketch is 2-3 orders of magnitude faster than the original Neo4j in terms of query performance. The P90 edge query latency is approximately 1μs with HourglassSketch, and over 60ms without; The P90 node query latency in Neo4j is also greatly reduced from 2ms to 10μs after adding HourglassSketch, demonstrating the superiority of HourglassSketch in modern graph databases.

VI. Conclusion

Graph stream storage is important in many fields. In this paper, we present a novel data structure, HourglassSketch, for accurate graph stream summarization. HourglassSketch uses two sketches, CocoSketch and TowerTCMSketch to store the graph, and the Error Funnel is added in the middle as optimization to reduce error. We give theoretical guarantees for HourglassSketch by strict derivation. Extensive experimental results show that HourglassSketch is faster, more accurate and more hardware-friendly.

Acknowledgment

We would like to thank the anonymous reviewers for their constructive comments. This work was supported in part by the National Key R&D Program of China (No. 2022YFB2901504), and in part by the National Natural Science Foundation of China (NSFC) (No. U20A20179, 62372009, 624B2005), research grant No. SH- 2024JK29, and High Performance Computing Platform of Peking University.

REFERENCES

[1] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

[2] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. Heterogeneous graph attention network. In *The world wide web conference*, pages 2022–2032, 2019.

[3] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph distances in the data-stream model. *SIAM Journal on Computing*, 38(5):1709–1727, 2009.

[4] Yuchen Zhao and Philip S Yu. On graph stream clustering with side information. In *Proceedings of the 2013 SIAM International Conference on Data Mining*, pages 139–150. SIAM, 2013.

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

[6] Hristo Djidjev, Gary Sandine, Curtis Storlie, and Scott Vander Wiel. Graph based statistical analysis of network traffic. In *Proceedings of the Ninth Workshop on Mining and Learning with Graphs*, 2011.

[7] Francesco Zola, Lander Segurola-Gil, Jan Lukas Bruse, Mikel Galar, and Raúl Orduna-Urrutia. Network traffic analysis through node behaviour classification: a graph-based approach with temporal dissection and data-level preprocessing. *Computers & Security*, 115:102632, 2022.

[8] Charu Aggarwal and Karthik Subbian. Evolutionary network analysis: A survey. *ACM Computing Surveys (CSUR)*, 47(1):1–36, 2014.

[9] Charu C Aggarwal. *An introduction to social network data analytics*. Springer, 2011.

[10] Yanxiang Huang, Bin Cui, Wenyu Zhang, Jie Jiang, and Ying Xu. Tencentrec: Real-time stream recommendation in practice. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 227–238, 2015.

[11] Youwei Wang, Weihui Dai, and Yufei Yuan. Website browsing aid: A navigation graph-based recommendation system. *Decision support systems*, 45(3):387–400, 2008.

[12] Zainab Abbas, Paolo Sottovia, Mohamad Al Hajj Hassan, Daniele Foroni, and Stefano Bortoli. Real-time traffic jam detection and congestion reduction using streaming graph analytics. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 3109–3118. IEEE, 2020.

[13] Zhishuai Li, Gang Xiong, Yonglin Tian, Yisheng Lv, Yuanyuan Chen, Pan Hui, and Xiang Su. A multi-stream feature fusion approach for traffic prediction. *IEEE transactions on intelligent transportation systems*, 23(2):1456–1466, 2020.

[14] Xu Chen, Junshan Wang, and Kunqing Xie. Trafficstream: A streaming traffic flow forecasting framework based on graph neural networks and continual learning. *arXiv preprint arXiv:2106.06273*, 2021.

[15] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. Towards large-scale graph stream processing platform. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 1321–1326, 2014.

[16] Hyunseok Seo, Jinwook Kim, and Min-Soo Kim. Gstream: A graph streaming processing method for large-scale graphs on gpus. *ACM SIGPLAN Notices*, 50(8):253–254, 2015.

[17] Ruijie Miao, Zheng Zhong, Jiarui Guo, Zikun Li, Tong Yang, and Bin Cui. Burstsketch: Finding bursts in data streams. *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[18] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data stream management: processing high-speed data streams*. Springer, 2016.

[19] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[20] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems (TOCS)*, 21(3):270–313, 2003.

[21] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[22] Kaicheng Yang, Sheng Long, Qilong Shi, Yuanpeng Li, Zirui Liu, Yuhan Wu, Tong Yang, and Zhengyi Jia. Sketchint: Empowering int with towersketch for per-flow per-switch measurement. *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[23] Zhuochen Fan, Ruixin Wang, Yalun Cai, Ruwen Zhang, Tong Yang, Yuhan Wu, Bin Cui, and Steve Uhlig. Onesketch: A generic and accurate sketch for data streams. *IEEE Transactions on Knowledge and Data Engineering*, 2023.

[24] Yikai Zhao, Wenchen Han, Zheng Zhong, Yinda Zhang, Tong Yang, and Bin Cui. Double-anonymous sketch: Achieving top-k-fairness for finding global top-k frequent items. *Proceedings of the ACM on Management of Data*, 1(1):1–26, 2023.

[25] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 741–756, 2018.

[26] Peiqing Chen, Dong Chen, Lingxiao Zheng, Jizhou Li, and Tong Yang. Out of many we are one: Measuring item batch with clock-sketch. In *Proceedings of the 2021 International Conference on Management of Data*, pages 261–273, 2021.

[27] Yuhan Wu, Shiqi Jiang, Yifei Xu, Siyuan Dong, Kaicheng Yang, Peiqing Chen, and Tong Yang. Unbiased real-time traffic sketching. *IEEE Transactions on Network Science and Engineering*, pages 1–13, 2023.

[28] Yinda Zhang, Peiqing Chen, and Zaoxing Liu. Octosketch: Enabling real-time, continuous network monitoring over multiple cores.

[29] Daniel Ting. Data sketches for disaggregated subset sum and frequent item estimation. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1129–1140, 2018.

[30] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: High-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 207–222, 2021.

[31] Nan Tang, Qing Chen, and Prasenjit Mitra. Graph stream summarization: From big bang to big crunch. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1481–1496, 2016.

[32] Zhiguo Jiang, Hanhua Chen, and Hai Jin. Auxo: A scalable and efficient graph stream summarization structure. *Proceedings of the VLDB Endowment*, 16(6):1386–1398, 2023.

[33] Xiangyang Gou, Lei Zou, Chenxingyu Zhao, and Tong Yang. Graph stream sketch: Summarizing graph streams with high speed and accuracy. *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[34] Arijit Khan and Charu Aggarwal. Query-friendly compression of graph streams. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 130–137. IEEE, 2016.

[35] Peixiang Zhao, Charu C Aggarwal, and Min Wang. gsketch: On query estimation in graph streams. *arXiv preprint arXiv:1111.7167*, 2011.

[36] Jihoon Ko, Yunbum Kook, and Kijung Shin. Incremental lossless graph summarization. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 317–327, 2020.

[37] Rui Qiu, Yi Ming, Yisen Hong, Haoyu Li, and Tong Yang. Windbell index: Towards ultra-fast edge query for graph databases. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 2090–2098. IEEE, 2023.

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

[39] Per Fuchs, Domagoj Margan, and Jana Giceva. Sortledton: a universal, transactional graph data structure. *Proceedings of the VLDB Endowment*, 15(6):1173–1186, 2022.

[40] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E Gonzalez, and Ion Stoica. {TEGRA}: Efficient {Ad-Hoc} analytics on evolving graphs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 337–355, 2021.

[41] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Commongraph: Graph analytics on evolving data. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 133–145, 2023.

[42] Zhuochen Fan, Yalun Cai, Zirui Liu, Jiarui Guo, Xin Fan, Tong Yang, and Bin Cui. Cuckoograph: A scalable and space-time efficient data structure for large-scale dynamic graphs. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 2025.

[43] Kai Cheng, Limin Xiang, and Mizuho Iwaihara. Time-decaying bloom filters for data streams with skewed distributions. In *15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications (RIDE-SDMA'05)*, pages 63–69. IEEE, 2005.

[44] Jiarui Guo, Yisen Hong, Yuhan Wu, Yunfei Liu, Tong Yang, and Bin Cui. Sketchpolymer: Estimate per-item tail quantile using one sketch. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 590–601, 2023.

[45] Haoyu Li, Qizhi Chen, Yixin Zhang, Tong Yang, and Bin Cui. Stingy sketch: a sketch framework for accurate and fast frequency estimation. *Proceedings of the VLDB Endowment*, 15(7):1426–1438, 2022.

[46] Peiqing Chen, Yuhan Wu, Tong Yang, Junchen Jiang, and Zaoxing Liu. Precise error estimation for sketch-based flow measurement. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 113–121, 2021.

[47] Yuhan Wu, Shiqi Jiang, Siyuan Dong, Zheng Zhong, Jiale Chen, Yutong Hu, Tong Yang, Steve Uhlig, and Bin Cui. Microscopesketch: Accurate sliding estimation using adaptive zooming. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 2660–2671, 2023.

[48] Xiangyang Gou, Long He, Yinda Zhang, Ke Wang, Xilai Liu, Tong Yang, Yi Wang, and Bin Cui. Sliding sketches: A framework using time zones for data stream processing in sliding windows. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1015–1025, 2020.

[49] Kostas Patroumpas and Timos Sellis. Window specification over data streams. In *International Conference on Extending Database Technology*, pages 445–464. Springer, 2006.

[50] Wang Bao-Jun and Zhan Ying. A survey and performance evaluation on sliding window for data stream. In *2011 IEEE 3rd International Conference on Communication Software and Networks*, pages 654–657. IEEE, 2011.

[51] Yikai Zhao, Yubo Zhang, Pu Yi, Tong Yang, Bin Cui, and Steve Uhlig. The stair sketch: Bringing more clarity to memorize recent events. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 164–177. IEEE, 2022.

[52] Corinna Vehlow, Fabian Beck, Patrick Auwärter, and Daniel Weiskopf. Visualizing the evolution of communities in dynamic graphs. In *Computer graphics forum*, volume 34, pages 277–288. Wiley Online Library, 2015.

[53] Timothy M Chan, Mihai Patraşcu, and Liam Roditty. Dynamic connectivity: Connecting to networks and geometry. *SIAM Journal on Computing*, 40(2):333–349, 2011.

[54] Brenden Lake and Joshua Tenenbaum. Discovering structure by learning sparse graphs. 2010.

[55] Michihiro Kuramochi and George Karypis. Finding frequent patterns in a large sparse graph. *Data mining and knowledge discovery*, 11(3):243–271, 2005.

[56] Andrea Montanari. Finding one community in a sparse graph. *Journal of Statistical Physics*, 161:273–299, 2015.

[57] The source code and technical report of HourglassSketch. https://github.com/HourglassSketch/HourglassSketch-code.

[58] Zhuochen Fan, Jiarui Guo, Xiaodong Li, Tong Yang, Yikai Zhao, Yuhan Wu, Bin Cui, Yanwei Xu, Steve Uhlig, and Gong Zhang. Finding simplex items in data streams. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 1953–1966. IEEE, 2023.

[59] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. Heavykeeper: an accurate algorithm for finding top-k elephant flows. *IEEE/ACM Transactions on Networking*, 27(5):1845–1858, 2019.

[60] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. Heavyguardian: Separate and guard hot items in data streams. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2584–2593, 2018.

[61] The CAIDA Anonymized Internet Traces. http://www.caida.org/data/overview/.

[62] The source code of Bob Hash. http://burtleburtle.net/bob/hash/evahash.html.

[63] The DBLP Archive. https://dblp.dagstuhl.de/xml/.

[64] The network dataset internet traces. http://snap.stanford.edu/data/.

[65] The Neo4j website. https://neo4j.com/.

[66] Barefoot tofino: World's fastest p4-programmable ethernet switch asics. https://barefootnetworks.com/products/brief-tofino/.