Auxo: A Scalable and Efficient Graph Stream Summarization Structure

Zhiguo Jiang
Huazhong University of Science and Technology
Wuhan, 430074, China
jiangzg@hust.edu.cn

Hanhua Chen
Huazhong University of Science and Technology
Wuhan, 430074, China
chen@hust.edu.cn

Hai Jin
Huazhong University of Science and Technology
Wuhan, 430074, China
hjin@hust.edu.cn

ABSTRACT

A graph stream refers to a continuous stream of edges, forming a huge and fast-evolving graph. The vast volume and high update speed of a graph stream bring stringent requirements for the data management structure, including sublinear space cost, computation-efficient operation support, and scalability of the structure. Existing designs summarize a graph stream by leveraging a hash-based compressed matrix and representing an edge using its fingerprint to achieve practical storage for a graph stream with a known upper bound of data volume. However, they fail to support the dynamically extending of graph streams.

In this paper, we propose Auxo, a scalable structure to support space/time efficient summarization of dynamic graph streams. Auxo is built on a proposed novel *prefix embedded tree* (PET) which leverages binary logarithmic search and common binary prefixes embedding to provide an efficient and scalable tree structure. PET reduces the item insert/query time from $O(|E|)$ to $O(log|E|)$ as well as reducing the total storage cost by a $log|E|$ scale, where $|E|$ is the size of the edge set in a graph stream. To further improve the memory utilization of PET during scaling, we propose a proportional PET structure that extends a higher level in a proportionally incremental style. We conduct comprehensive experiments on large-scale real-world datasets to evaluate the performance of this design. Results show that Auxo significantly reduces the insert and query time by one to two orders of magnitude compared to the state of the arts. Meanwhile, Auxo achieves efficiently and economically structure scaling with an average memory utilization of over 80%.

PVLDB Reference Format:
Zhiguo Jiang, Hanhua Chen, and Hai Jin. Auxo: A Scalable and Efficient Graph Stream Summarization Structure. PVLDB, 16(6): 1386 - 1398, 2023.
doi:10.14778/3583140.3583154

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CGCL-codes/Auxo.

[*]All the authors are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China. Hanhua Chen is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 16, No. 6 ISSN 2150-8097.
doi:10.14778/3583140.3583154

1 INTRODUCTION

A graph stream [8, 11, 12, 21, 25, 26, 34] represents a continuous sequence of items, where each item is defined as a triplet $e_i = (< s_i, d_i >; w_i; t_i)$, indicating an edge $s_i \rightarrow d_i$ with a weight value w_i appears at time instance t_i. A specific edge can repeatedly occur at different time instants with varying weights, and we can accumulate the weight values. The continuously coming edges form a dynamically extending graph $G(V, E)$, representing continuing interconnections or interactions among entities [6]. The graph stream data model can be widely used in emerging big data applications such as close contact identification in the anti-virus campaign [10], financial fraud detection in transaction systems [21], and user-behavior graph analysis [13].

However, storing a graph stream is a challenging issue. First, the data volume of a graph stream in real-world applications can be extraordinarily huge. Hence, a storage scheme should incur only sublinear storage costs. For example, Tencent's health code scan [35] can generate more than one billion records every day, making long-term COVID-19 spreading pattern analysis difficult. Large ISPs can transmit 10^9 packets per hour per router [14], raising great challenges for cyber security monitoring. It is clear that exactly storing such huge graph stream data is difficult. Second, to cope with a continuously produced graph stream, a storage scheme should be able to scale with the increase of the dataset. It is infeasible for a system with a pre-defined capacity to store a graph stream with a dynamically increasing volume.

Graph stream summarization structures have attracted many recent research efforts [2, 7, 8, 12, 18, 22, 34]. Existing designs can be classified into two types, hash-based and MDL (*Minimum Description Length*) based (MDL [28] states that given a model family MF and the data D needs to be compressed, the best model $M \in MF$ minimizes $L(M) + L(D|M)$, where $L(M)$ and $L(D|M)$ indicate the length to describe M and the length of the encoded data D, respectively.)

Hash-based graph stream summarizations represent the original graph stream with a hash-based compressed matrix and denote the items using their Boolean labels, achieving an approximate and practical storage scheme with sublinear memory cost. Tang *et al.* propose TCM [34], which leverages an $m \times m$ size preset compressed matrix M to store the items of a graph stream. It uses a hash function $h(\cdot)$ with range value $[0, m)$ to map an item of a graph stream to a corresponding bucket in M. For a coming item of edge $s_i \rightarrow d_i$, TCM adds its associated weight w_i to the value stored in the bucket $M[h(s_i), h(d_i)]$ (Nodes with the same hash value are merged). TCM can support both boolean queries (e.g., whether there exists an edge $x \rightarrow y$) and aggregation queries (e.g.,

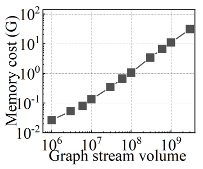


Figure 1: Memory cost increase

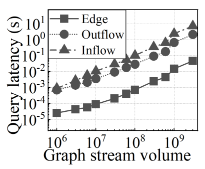


Figure 2: Query latency increase

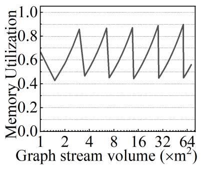


Figure 3: Utilization fluctuation

returning the aggregated weight of all the edges having x as the source node by adding up the weight values of the $h(x)_{th}$ column of M). However, TCM suffers from poor query accuracy due to hashing collisions. To improve accuracy, Gou *et al.* propose GSS [12], which stores the weight value of an inserted edge into the associated bucket with its fingerprint. When an occupied bucket encounters another coming edge with a different fingerprint, GSS stores its weight and fingerprint in an extra buffer out of M. GSS further relies on a *square hashing* scheme to allocate multiple candidate buckets in M for each edge to reduce the size of the extra buffer. In a word, although existing hash-based size-predefined graph stream summarization designs achieve sublinear storage cost and fast edge updating speed, their structures do not scale facing real-world graph streams where edges arrive continuously with unknown bounds.

MDL-based graph stream summarizations, such as MoSSo [18] and SGS [22], utilize MDL to measure the space cost of the summarized graph and move nodes between supernodes to reduce MDL as every edge arrives. Those structures are scalable but inefficient, especially for fast-updating and large-scale graph streams. First, MDL-based strategies are much more computation costly than the hash-based scheme. Second, they still use traditional data structures like adjacency list to store the summarized graph, which may bring prohibitively high memory cost.

All in all, existing graph stream summarization structures are either with poor scalability or time/space inefficient when facing large-scale and fast-updating graph streams. How to design an efficient and scalable graph stream summarization scheme is a nontrivial and unsolved problem.

A straightforward way to extend a hash-based graph stream summarization structure is to extend a new building block of compressed matrix whenever the structure is full. However, such a scheme raises linearly increasing computation and memory costs when the graph stream scales. The overhead can be prohibitively high for extraordinarily large-scale datasets, making the structure unscalable in practice. In Figures. 1 and 2, we conduct experiments to evaluate the performance of GSS [12], the state-of-the-art graph stream summarization structure, following the linearly expanding scheme. In the experiment, when an initial GSS compressed matrix M_0 is full (i.e., all the candidate buckets of an inserting edge are occupied), we generate a new empty building block (i.e., a new homogeneous compressed matrix M_1) link it to M_0. Newly coming edges are inserted into M_1 until M_1 is full and M_2 is created. In this way, we achieve a chain-style structure based on GSS (we call it GSS_Chain for short). In the experiment, we use the dataset of the hyperlink network in the UK domain for the United Kingdom (2007) [3], which contains 105 million nodes and 3.3 billion edges. Figures 1 and 2 plot the memory cost and the query latency for different queries over GSS_Chain as the edges are inserted. Figure 1 shows the memory cost of GSS_Chain increases linearly with the volume of a graph stream. Figure 2 plots latency for different queries. We can see GSS_Chain needs nearly ten seconds to process a node-in flow query (obtaining the aggregated weight of a node's in-going edges). The results reveal that a chain-style extending structure can raise prohibitively high memory and computation costs in the presence of large-scale graph streams.

To solve the problem, we propose Auxo, a scalable and efficient structure for graph stream summarization. Two factors contribute to the efficiency of Auxo. First, Auxo proposes a novel *prefix embedded tree* (PET) which extends new building blocks in a tree-style to achieve logarithmic computation cost for insert/query processing. Moreover, PET embeds the prefix information inside the tree. Thus, in all the extended building blocks on the ith level of the tree, Auxo can omit an i-bit prefix for every inserted fingerprint without sacrificing query accuracy. Based on PET, Auxo achieves a scalable structure and reduces the insert/query time from $O(|E|)$ to $O(log|E|)$ as well as saving the memory cost by $\frac{log_2 n-2}{b}$, where n is the number of the storage block and b is the size of a storage cell.

Second, to improve the memory utilization of PET, we propose a proportionally incremental strategy to expand PET by exploiting the principle of proportional sequence. Figure 3 shows that the memory utilization of basic PET can fluctuate at every moment when expanding a new level for the increasing graph stream. Our proposed incremental expanding scheme decomposes the exponentially expanding of a new level into a proportionally incremental sequence, avoiding the under-utilization of memory.

We conduct comprehensive experiments on large-scale real-world datasets to evaluate this design. The results show that Auxo reduces the time of insert and query by one to two orders of magnitude compared to state-of-the-art designs.

All in all, the contributions of this work are threefold:

- We propose a novel *prefix embedded tree* (PET), with which we design a scalable graph stream summarization structure called Auxo. Auxo achieves logarithmic time for item insert/query as well as saving $\frac{log_2 n-2}{b}$ of the storage space.
- We further propose a novel proportionally incremental strategy for PET by exploiting the principle of proportional sequence and the strategy greatly improves the memory utilization of Auxo.
- Experimental results show that Auxo significantly reduces insert/query time as well as improves memory utilization compared to state-of-the-art designs.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents the design of Auxo. Section 4 theoretically analyzes the structure. Section 5 evaluates the performance. Section 6 concludes this work and discusses future work.

2 RELATED WORK

Graph stream summarization [2, 7, 8, 12, 18, 22, 34] have attracted many recent research efforts. According to the node merging strategy, existing designs can be classified into two categories, hash-based [7, 8, 12, 34] and MDL-based [18, 22].

Hash-based graph stream summarizations represent items of a graph stream by their fingerprints and leverage a hash-based compressed matrix for storage, achieving memory/computation

Table 1: Comparison of existing work

Data structure	Main idea	Hash based	Lossy	Time, Space cost	Accuracy guarantee	Scalable
TCM	Compressed matrix	✓	✓	$O(1), O(\|E\|)$	$e^{-\frac{\|E\|}{m^2}}$	×
GSS	Fingerprint reducing error	✓	✓	$O(1), O(\|E\|)$	$e^{-\frac{\|E\|}{4^f m^2}}$	×
Scube	Dynamiclly space allocating	✓	✓	$O(1), O(\|E\|)$	$e^{-\frac{\|E\|}{4^f m^2}}$	×
Horae	Range binary decomposition	✓	✓	$O(log\|T\|), O(\|E\|log\|T\|)$	$e^{-\frac{\|E\|}{4^f m^2}}$	"✓"
MoSSo	Moving nodes reducing MDL	×	×	$O(\sum deg(v)), O(\|E\|)$	\	✓
SGS	Constraining affected nodes	×	×	$O(\sum deg(v)), O(\|E\|)$	\	✓
GS4	Similarity based node clustering	×	✓	$O(\|V\|^3), O(\|E\|)$	×	✓
Auxo	Prefix embedded tree	✓	✓	$O(log\|E\|), O(\|E\|(1 - log\|E\|))$	$e^{-\frac{\|E\|}{4^f m^2}}$	✓

efficient approximate graph stream management with slight accuracy loss. Tang *et al.* propose TCM [34], which sketches a graph stream using an $m \times m$ hash-based compressed matrix M to achieve approximate graph stream management with sublinear memory cost. Specifically, for an edge $s_i \longrightarrow d_i$, TCM maps s_i and d_i with a hash function $h(\cdot)$ into the range $[0, m)$ and adds the weight value associated with e_i (i. e., w_i) to the value stored in the bucket $M[h(s_i), h(d_i)]$. TCM can support boolean queries as well as aggregation queries. For example, one can obtain the accumulated weight of edge $s_i \longrightarrow d_i$ by returning the value stored in $M[h(s_i), h(d_i)]$. The result can be over-estimated due to hash collision. Dmatrix [17] integrates TCM with the representative key reservation and majority voting technology to improve accuracy. To manage graph streams whose nodes and edges have labels, SBG [16] allocates different compressed matrices to store edges with different labels and designs a priority rule and evicting strategy to reduce hash collision. Song *et al.* propose LGS [32], which divides the compressed matrix into multiple areas to indicate the node labels and utilizes prime numbers to indicate the edge labels. However, such schemes are variants of TCM and thus suffer from poor query accuracy.

To address the problem of accuracy in TCM, Gou *et al.* propose GSS [12], which stores an f-bit fingerprint ξ_v (i.e., a hash value) for each node v with the weight in the compressed matrix. Formally, GSS consists of an $m \times m$ compressed matrix M and an adjacency list as an extra buffer. When inserting an item $e_i = (< s_i, d_i >; w_i; t_i)$, GSS respectively computes the fingerprints for s_i and d_i (i.e., $< \xi_{s_i}, \xi_{d_i} >$), and stores them together with w_i into $M[h(s_i), h(d_i)]$. If two edges with different fingerprint pairs collide in the same bucket, GSS stores the newcomer in the buffer. To avoid a large scale buffer, GSS improves the utilization of M by generating a sequence of hash addresses $\{h_1(s_i/d_i), h_2(s_i/d_i), ..., h_r(s_i/d_i)\}$ from the original hash address $h(s_i/d_i)$ using linear congruence method [20]. Then, a sample of buckets $\{M[h_k(s_i), h_j(d_i)] | 1 \leq k \leq r,\ 1 \leq j \leq r\}$ is chosen as the candidate buckets for $s_i \longrightarrow d_i$. GSS records $< \xi_{s_i}, \xi_{d_i}, w_i, idxpair >$ in an empty candidate bucket, where $idxpair$ represents the indexes of the candidate bucket (for $M[h_k(s_i), h_j(d_i)]$, $idxpair = (k, j)$). However, if all the candidate buckets are occupied, GSS inserts the leftover edges into the buffer. For a large-scale graph stream with edges arriving continuously and volume unpredictably, linearly increasing large buffer causes a prohibitively memory cost and insertion latency.

To support efficient temporal range queries for graph streams (e.g., whether the edge $x \longrightarrow y$ appeared between temporal range $[t, t + L)$), Chen *et al.* propose Horae [8] which maintains $log|T|$ identical compressed matrices with different time granularity where $|T|$ denotes the length of the temporal range. When inserting an edge, Horae embeds the prefix information of the binary timestamp into the fingerprint to indicate the time granularity (Horae's way of embedding is using the concatenation of the node label and the prefix of the binary timestamp to generate the fingerprint). Thus, an edge should be inserted into all the $log|T|$ compressed matrices, and for a compressed matrix with coarser time granularity, a shorter prefix is embedded. When evaluating a temporal range query with length L, Horae decomposes the temporal range into at most $logL$ subranges with different granularity and queries the result from the compressed matrix with the corresponding time granularity. As T increases, Horae builds compressed matrices with coarser time granularity to maintain the efficiency of the temporal range queries and copies the whole previously inserted graph stream into the newly created matrix for data integrity. However, Horae is still an unscalable structure from the perspective of graph stream volume increasing, while the purpose of Horae's scaling is to maintain the efficiency of the time range queries. Horae maintains $log|T|$ compressed matrices and inserts every single edge into each compressed matrix. However, each compressed matrix is still unscalable with edges inserted continuously. Therefore, the space cost of Horae is $|E|log|T|$ (the space cost of Auxo is $|E|(1 - log|E|)$). For insertion, Horae has a time cost of $log|T|$ (the time cost of Auxo is $log|E|$). To efficiently manage graph stream summarization with skewed node degree, Chen *et al.* propose Scube [7] which designs an efficient probabilistic counting scheme for identifying high-degree nodes and allocates more buckets for them.

MDL-based graph stream summarization schemes utilize MDL to measure the memory cost of the summarized graph and merge nodes with heuristic strategies to reduce the space cost. Ko *et al.* propose MoSSo [18] for lossless graph stream summarization. MoSSo summarizes the original graph into a graph of supernodes ($L(M)$ in MDL) and two edge correction sets for recovery ($L(D|M)$ in MDL). For each edge update, MoSSo moves the nodes between supernodes to reduce MDL. To improve the summarization quality caused by parameter selection of MoSSo, Ma *et al.* propose SGS [22], which constrains the affected nodes within the 2-hop neighbors of the inserted nodes. Ashraf Payaman *et al.* propose GS4 [2] to summarize a graph stream by node clustering based on both the structure and semantics in a time window and update the summarization if the results are significantly different between two consecutive windows. Compared with the hash-based strategy, MDL-based node merging strategies still utilize traditional data structures like adjacency list to store the summarized result which may bring prohibitively high memory costs and take much more computing cost for updating an edge when facing a large-scale graph stream. Table 1 summarizes recent efforts of graph stream summarization from the perspectives of the main idea, time/space costs, accuracy, scalability, etc. To the best of our knowledge, Auxo is the first scalable graph stream summarization structure with both time and space efficiency.

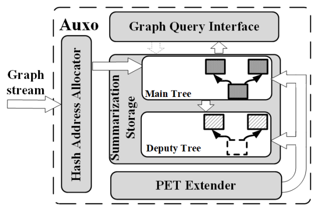


Figure 4: Architecture of Auxo

Other related work focuses on dynamic graph summarization. Shah *et al.* propose TimeCrunch [29] to find coherent, temporal patterns in a dynamic graph and utilize MDL to minimize the memory cost. Tsalouchidou *et al.* [36] treat the adjacency matrix of each timestamp as a tensor and summarize the tensors in the current time window by clustering and update the summarized result in an incremental way. Pensieve [37] proposes a skewness-aware multi-version dynamic graph management system, which leverages a differentiated storage scheme to cope with high degree vertices and low degree vertices separately. Other efforts manage graph streams for specific applications such as event detection [1, 21, 25, 27] and monitoring the characteristic attributes of graph streams [11, 30, 33].

3 AUXO DESIGN

Figure 4 shows the architecture of Auxo which contains a hash address allocator, a summarization storage structure, a PET extender, and the graph query interface. Whenever inserting an edge, the hash address allocator generates the addresses of the candidate buckets and the fingerprints of the edge. Then, Auxo queries the Main tree (a PET) to find whether it has stored the edge before. If not, it inserts the edge into the Deputy tree (an auxiliary structure helps the Main tree to extend a new level in a proportional way). Once the building block on the Deputy tree are full, the PET extender extends it to scale the storage structure. Furthermore, if the Deputy tree has been generated to the next level of the Main tree, it relinks its last level to the Main tree. The graph query interface integrates graph-related queries for various applications.

3.1 Overview

In this section, we first introduce the overview of Auxo and then present the detailed design. Table 2 lists the notations we use. To avoid the linearly increasing space/time costs in a chain-style scaling structure, Auxo explores a computation and memory efficient scheme for scaling the graph stream summarization. Specifically, we propose a novel *prefix embedded tree* (PET) which improves the efficiency of Auxo in two aspects: 1) PET omits a common binary prefix of fingerprints by embedding the prefix information inside a tree structure, and thus saves a significant amount of space costs; 2) PET leverages a logarithmic binary searching algorithm to reduce the computation costs for item insert and query. Moreover, we design a proportionally incremental strategy for PET to improve Auxo's memory utilization, avoiding exponentially expanding.

Figure 5 illustrates our basic idea of PET using an example with the GSS compressed matrix [12] as a building block. In the example, the fingerprint length is five and the structure has two layers. The first layer consists of one compressed matrix M_0 which stores the pair of original five-bit fingerprints for an inserted edge. When M_0

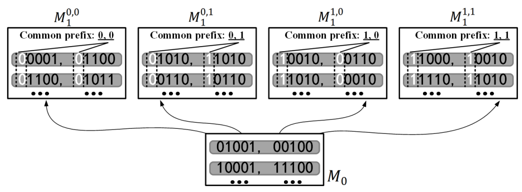


Figure 5: An example with insight about PET

is full, the structure extends to the second layer, which includes four compressed metrics: $M_1^{0,0}$, $M_1^{0,1}$, $M_1^{1,0}$, and $M_1^{1,1}$. When inserting a newly coming edge $s_i \rightarrow d_i$ into the second layer, we first examine the one-bit prefixes of the fingerprints ξ_{s_i} and ξ_{d_i}. There are four cases including (0,0), (0, 1), (1, 0), and (1, 1). Our insight here is that if the first bit prefixes of ξ_{s_i} and ξ_{d_i} are 0 and 1 respectively, we can insert the fingerprints into $M_1^{0,1}$ with the first bit prefixes omitted. Hence, the lengths of the fingerprints stored in the second layer are reduced from five to four. Similarly, if ξ_{s_i} and ξ_{d_i} both start with the prefix "1", we can insert ξ_{s_i} and ξ_{d_i} into $M_1^{1,1}$ with the first one-bit prefix omitted. In the same way, we can save the two-bit prefixes of the fingerprints of the edges inserted into the third layer. Generally, on layer i, we can omit the i-bit prefixes of the fingerprints of the inserted edges. Therefore, we can save a significant amount of storage costs by embedding the prefix information in the tree structure of PET. When querying an edge, we need to check only one matrix on each level of PET according to its pair of prefixes. Thus, the time cost for a query is in the logarithmic scale.

3.2 Prefix Embedded Tree

Prefix embedded tree (PET) aims at providing a scalable summarization framework for dynamically increasing graph streams. Given a graph stream consisting of continuously coming edge items $\{(<s_i, d_i>; w_i; t_i)\}$, we define the prefix embedded tree as follows.

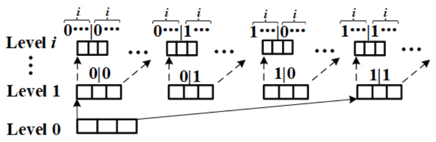
Definition 1 (Prefix Embedded Tree). *For a graph stream summarization structure which identifies an edge $s_i \rightarrow d_i$ using a pair of fingerprints <ξ_{s_i}, ξ_{d_i}>, the prefix embedded tree scales the storage building blocks in a 4-ary tree which has the following features.*

Feature A: On level l, PET has a number of 4^l storage building blocks $\{block_l^0, ..., block_l^{4^l-1}\}$. Each building block consists of a set of buckets and each bucket records the fingerprints pair of an edge.

Feature B: On level l, an edge with fingerprint pairs ξ_{s_i} and ξ_{d_i} is stored in $block_l^{(\xi_{s_i}^l | \xi_{d_i}^l)}$, where ξ_v^l represents the l-bit prefixes of ξ_v while the notation $'|'$ denotes the concatenation operation. For example, on level 2, an edge with a pair of fingerprints ($\underline{01}101$, $\underline{10}011$) is stored in $block_2^{01|10}$ (also denoted as $block_2^{0110}$ or $block_2^6$).

Feature C: On level l, PET stores an edge's fingerprint pairs with their l-bit prefixes omitted, denoted as $< \xi_{s_i}^{-l}, \xi_{d_i}^{-l} >$. In the above example, PET stores the prefixes omitted fingerprint pair (101, 011) of the inserted edge in $block_2^{01|10}$.

Figure 6 illustrates the PET structure. The level i of the PET structure consists of 4^i storage building blocks. On level 0, PET contains only one storage building block. When PET extends to level one, it expands four storage building blocks, i.e., $block_1^{0|0}$, $block_1^{0|1}$, $block_1^{1|0}$, $block_1^{1|1}$ corresponding to the four cases of $(\xi_{s_i}^1 | \xi_{d_i}^1)$. On level one, PET can omit the one-bit prefixes of the inserted fingerprints. Similarly, when PET extends to level i, it expands 4^i



Level i
Level 1
Level 0
$\overbrace{0\cdots}^{i}|\overbrace{0\cdots}^{i}$
$\overbrace{0\cdots}^{i}|\overbrace{1\cdots}^{i}$
$\overbrace{1\cdots}^{i}|\overbrace{0\cdots}^{i}$
$\overbrace{1\cdots}^{i}|\overbrace{1\cdots}^{i}$
0|0
0|1
1|0
1|1

Figure 6: Structure of PET

storage building blocks and omits the i-bit prefixes of the inserted fingerprints. Formally, we have the following theorems about PET.

Theorem 1. *An l-level PET reduces the memory cost by the ratio with a lower bound of $\frac{l-4/3}{f}$, where f is the length of a fingerprint.*

Proof. Supposing a storage building block has a number of σ buckets, without omitting the prefix, the total memory cost of the l levels of homogeneous storage building blocks is:

$$M_o = \sum_{i=0}^{l-1} 2f\sigma 4^i = 2\sum_{i=0}^{l-1} 4^i f\sigma \tag{1}$$

With the proposed PET design, the total amount of saved memory cost is computed by:

$$M_s = \sum_{i=0}^{l-1} 2i\sigma 4^i = 2\sum_{i=0}^{l-1} 4^i i\sigma \tag{2}$$

Therefore, PET saves the memory cost by a ratio of M_s/M_o:

$$\frac{M_s}{M_o} = \frac{4^l(l-4/3)+4/3}{f(4^l-1)} > \frac{l-4/3}{f} \tag{3}$$

Theorem 1 is thus proved.

According to the 4-ary tree structure, an l–level PET has a total number of $n = \frac{4^l-1}{3}$ storage building blocks. In real applications, the number of building blocks n is proportional to the size of the edge set $|E|$. By simply varying Formula (3), we can see that PET can save the memory cost by a $log_4|E|$ scale. (We will give a more comprehensive analysis in Section 4.3)

Theorem 2. *Querying an edge with PET needs $O(log_4|E|)$ time, where $|E|$ is the size of the edge set.*

Proof. When querying an edge $x \longrightarrow y$, PET first computes the fingerprints ξ_x and ξ_y. Then, PET checks the fingerprints against the structure level by level from the root to the leaf to find the matched result. On level i, PET uses the i-bit prefixes of ξ_x and ξ_y to locate the storage building block possibly hosting the edge. Specifically, on level 0, PET checks $block_0$; on level 1, it checks $block_1^{(\xi_x^1|\xi_y^1)}$; and generally on level i, it checks $block_i^{(\xi_x^i|\xi_y^i)}$. For a l-level PET, it checks l blocks in the worst case. As aforementioned, the time cost for querying an edge in a single building block is constant by using a compressed matrix as a building block. At the same time, the total number of building blocks on PET increases linearly with $|E|$. Therefore, the PET's 4-ary tree needs $O(log_4|E|)$ time for edge query processing. Theorem 2 is thus proved.

PET can naturally support a graph stream to extend to a more general definition with a number of k ($k > 2$) identifying Boolean labels. Accordingly, we define a k-way PET (i.e., PET-k) which scales in a 2^k-ary tree. Formally, a PET-k with l levels of storage building blocks saves the memory cost by a ratio of $\frac{l-1-1/(2^k-1)}{f}$, while it needs $O(log_{2^k}|E|)$ time for querying an edge. Table 3 shows the time and memory costs reduced by PET-k with an example of parameter settings $l = 12$ and $f = 16$. We can see that PET achieves significant computation and memory reductions.

Table 3: Time and memory costs reduced by PET-k

Number of Boolean labels	$k = 2$	$k = 3$	$k = 4$	$k = 5$
Time reduced ratio	$1-10^{-6}$	$1-10^{-9}$	$1-10^{-12}$	$1-10^{-16}$
Memory reduced ratio	66.7%	67.9%	68.3%	68.5%

Table 2: Notations in Auxo

Notations	Description
$G(V, E)$	the graph stream to be represented with V and E indicating the vertices set and the edge set
$\lvert V\rvert$	size of the vertices set
$\lvert E\rvert$	size of the edge set
$e_i = (< s_i, d_i >; w_i; t_i)$	the item of edge $s_i \to d_i$ with weight w_i and timestamp t_i
f	length of the fingerprint
b	size of the bucket on level 0 in Auxo
$h(v)$	the original hash address of node v
$< \xi_{s_i}, \xi_{d_i} >$	fingerprint pair of edge $s_i \to d_i$
m	side width of the compressed matrix
r	length of the hash address sequence
$\{h_k(v)\lvert 1 \le k \le r\}$	hash address sequence of node v
p	number of the candidate buckets
l	number of levels in Auxo
n	number of compressed matrices allocated
α	the load factor of the compressed matrix
ξ_v^{-i}	fingerprint ξ_v without the i-bit prefix
$M_l^{x,y}$	the compressed matrix storing the edges with fingerprint prefix x, y for ξ_{s_i} and ξ_{d_i} on level l

3.3 Proportionally Incremental Strategy

Theorems 1 and 2 show PET's memory/computation efficiency. However, the exponentially scaling of the basic PET can have unsatisfied memory utilization whenever initializing a new level due to the large fraction of unused buckets on the new level. Today, many real-world applications deploy services across the cloud. According to the report on Cloud Adoption [23] in 2020, more than 88% of enterprises use the cloud in one form or another. For cloud services, service providers typically charge based on the amount of resources allocated and the length of service time. Many recent efforts [15, 24, 31] focus on overcoming the challenge of reducing resource usage cost while guaranteeing service quality. To improve the memory efficiency of PET, we design a lazy expanding strategy for PET that supports proportionally incremental extending.

The idea of the lazy expanding strategy is to make a proper trade-off between memory utilization and time cost. Specifically, the lazy expanding strategy organizes the building blocks of PET into a binary tree structure and alternately embeds the prefixes of ξ_{d_i} and ξ_{s_i}. A binary tree based structure achieves at least 50% memory utilization in the worst case while still constraining the query time in a logarithmic scale. Moreover, PET's lazy expanding strategy further explores proportionally incremental expanding by exploiting the feature of the proportional sequence. Specifically, we elaborate an auxiliary structure called *Deputy tree* to convert the exponentially expanding of a new level into a proportionally incremental sequence. We call the binary tree structure of the lazy expanding PET the Main tree. We find that for the Main tree, the numbers of building blocks vary as the sequence {1, 2, 4, ...} from the root to the leaf level, forming a proportion sequence. Hence, a Deputy tree can gradually construct the level i from level 1.

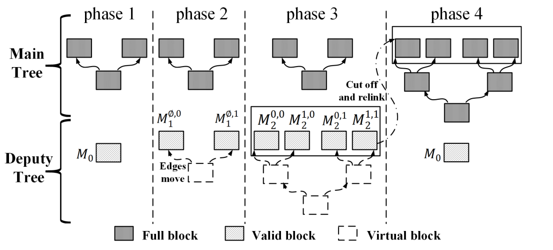


Figure 7: How the deputy tree extends a new level

Figure 7 shows how the lazy expanding PET uses the Deputy tree to expand to level two. We use the notation $M_l^{x,y}$ to indicate the compressed matrix based building block which stores the edge with fingerprint prefix $< x, y >$ for $< \xi_{s_i}, \xi_{d_i} >$ on level l (for simplicity, in the following, we use the term "matrix" to stand for "compressed matrix based building block" for short). The procedure includes four phases. In phase 1, PET generates only one matrix M_0 on the Deputy tree and inserts the upcoming edges into M_0. Once M_0 is full, PET generates two new matrices $M_1^{\varnothing,0}$, $M_1^{\varnothing,1}$ on level 1 for the Deputy tree (here, the notation $\varnothing$ indicates no prefix is embedded). Then, PET moves the items stored in M_0 to $M_1^{\varnothing,0}$ and $M_1^{\varnothing,1}$ according to our prefix embedding scheme. After that, PET releases the memory of M_0 and keeps the pointers as phase 2 shows. When level 1 is full, PET performs the same edge moving and memory release operations in phase 3. When level 2 is full, PET cuts off the level 2 of the Deputy tree and relinks it to the Main tree as shown in phase 4. After that, PET constructs a new Deputy tree with only one matrix M_0 for extending the Main tree's next level.

The edge-moving operation is efficient because it is performed with contiguous memory space and without edge relocation. When extending to a higher level, the proportionally incremental style of the Deputy tree achieves much better memory efficiency than exponentially extending and we have the following Theorem 3.

THEOREM 3. *The lowest bucket utilization of a lazy expanding PET with the proportionally incremental strategy is* 0.75.

PROOF. Considering a lazy expanding PET with l levels (totally $2^l - 1$ matrices) on the Main tree, when extending to the $(l+1)_{th}$ level, the number of matrices on the Deputy tree gradually increases as a proportional sequence $\{2^0, 2^1, ..., 2^l\}$. It is not difficult to see that the lowest bucket utilization happens when the Deputy tree extends to a higher level, which is

$$\begin{aligned} LBU &= \min\{1 - \frac{2^{i-1}}{2^l - 1 + 2^i}\}\ i = 1, 2, ..., l \\ &\approx 0.75 \end{aligned} \tag{4}$$

Theorem 3 is thus proved.

Figure 8 shows that the lowest bucket utilization of a lazy expanding PET with the proportionally incremental strategy is 0.75, which happens when a Deputy tree extends its last level. Figures 9 and 10 plot the cumulative distribution function and the average memory utilization of PET, lazy expanding PET, and lazy expanding PET with proportionally incremental strategy, respectively. We can see that the lazy expanding strategy improves the average memory utilization by 52% to 74% while the proportionally incremental strategy improves the average memory utilization by 74% to 93%.

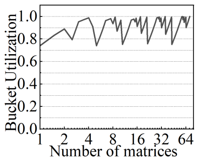


Figure 8: Bucket utilization

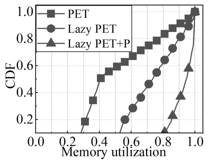


Figure 9: Memory utilization CDF

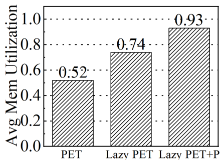


Figure 10: Average memory utilization

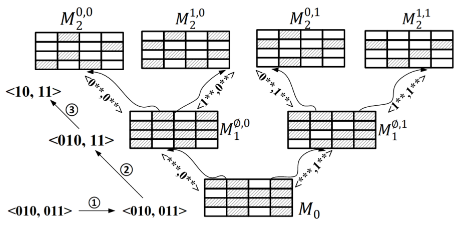


Figure 11: Basic Auxo structure

3.4 Auxo Structure Based on PET

A basic Auxo takes the advantage of the lazy expanding strategy. Figure 11 shows an example of a basic Auxo.(the shaded buckets indicate occupied while they are randomly distributed in the matrices). Initially, Auxo has only one matrix M_0 on level 0. For inserting an edge, Auxo searches for an available empty candidate bucket in M_0. If succeed, Auxo inserts $< \xi_{s_i}, \xi_{d_i}, w_i, idxpair >$ into M_0. Otherwise, Auxo extends to level 1, which contains two matrices $M_1^{\varnothing,0}$ and $M_1^{\varnothing,1}$. Then, it checks the first bit of fingerprint ξ_{d_i}. If ξ_{d_i} starts with "0", Auxo inserts $< \xi_{s_i}, \xi_{d_i}^{-1}, w_i, idxpair >$ into $M_1^{\varnothing,0}$ (ξ_v^{-i} denotes the fingerprint ξ_v with the i-bit prefix omitted); otherwise, it inserts the same record into $M_1^{\varnothing,1}$. When all the matrices on level 1 are full, Auxo extends to level 2 with matrices $M_2^{0,0}$, $M_2^{1,0}$, $M_2^{0,1}$, and $M_2^{1,1}$. Then, it checks the first bits of ξ_{d_i} and ξ_{s_i} to determine the matrix for inserting. Generally, whenever Auxo scales, it generates two children matrices on a higher level for each matrix on the current level and embeds the next bit of the prefixes of fingerprints ξ_{s_i} and ξ_{d_i} alternately on the new level.

In Figure 11, we show an example of item insertion in Auxo. Assume the edge has a fingerprint pair $< \xi_{s_i}, \xi_{d_i} >=< 010, 011 >$. First, Auxo traverses the candidate buckets in M_0 with fingerprint pair $< 010, 011 >$. If the insertion fails, as the fingerprint "011" starts with "0", Auxo checks $M_1^{\varnothing,0}$ with the fingerprint pair $< 010, 11 >$, and then checks $M_2^{0,0}$ with fingerprint pair $< 10, 11 >$. If the insertion in $M_2^{0,0}$ fails, Auxo extends to a higher level and inserts the edge into $M_3^{0,01}$ with fingerprint pair $< 10, 1 >$. Naturally, the proportionally incremental strategy stated in Section 3.3 can be applied to our basic Auxo design to improve memory utilization. Thus, we achieve a design of the proportional Auxo. The extending scheme of proportional Auxo is the same as Figure 7 shows.

3.5 Operations of Auxo

In this section, we present the operations of Auxo, including insert, extend, edge query, and node query.

Insert. Algorithm 1 presents the operations of the insert. First, Auxo checks the matrices on the Main tree from the root to a leaf level by level according to the fingerprint prefix (lines 4-10). If the Main tree has no matched item, Auxo further traverses the Deputy tree (lines 11-28) from the root to the leaf until we reach *CurM*, the associated matrix on the leaf level. Then we check whether any item matches in *CurM*. If Auxo finds a matched item, it accumulates the weight value of the edge. Otherwise, it looks for an empty bucket for inserting. Once the insertion in *CurM* fails, if the Deputy tree has expanded to the next level of the Main tree, Auxo cuts off the leaf level of the Deputy tree and relinks it to the Main tree. Then, Auxo constructs a new Deputy tree for insertion (lines 21-25). Otherwise, it extends to the Deputy tree's next level (line 27).

Figure 12 illustrates a running example for inserting an edge $s_i \rightarrow d_i$ with the weight value w_i on the Main tree. Auxo first computes fingerprint pair $F = <\xi_{s_i}, \xi_{d_i}> = <101, 110>$ and the hash address sequence ($hq(s_i) = \{0, 2\}$, $hq(d_i) = \{0, 3\}$). Then, Auxo checks M_0 first. If no items in M_0 match, Auxo continues to check $M_1^{\emptyset,1}$ as $\xi_{d_i} = 110$ starts with "1" and then checks $M_2^{1,1}$ as $\xi_{s_i} = 101$ starts with "1". Figure 12 also shows the detailed operations of Auxo on $M_1^{\emptyset,1}$. Auxo cuts off the first bit of the ξ_{d_i} as Auxo embeds it on level one. Then, Auxo checks the two candidate buckets $M[0, 0]$ and $M[2, 3]$. Auxo checks $M[0, 0]$ and finds that the index pair $I =< 1, 1 >$ matches while the cut fingerprint pair $F =< 101, 00 >$ does not match. Next, Auxo checks $M[2, 3]$ and finds the fingerprint pair and the index pair both match those stored in the bucket. This means Auxo has inserted this edge before and therefore it accumulates the weight w_i.

Algorithm 1: Insert($(< s_i, d_i >; w_i; t_i)$)

```
   Data: Edge $s_i \rightarrow d_i$, with weight $w_i$, timestamp $t_i$
1  $CandiAddresses,\ IndexPairs,\ \xi_{s_i},\ \xi_{d_i} \leftarrow$
     $getInsertInfo(s_i, d_i)$;
2  $MainLevel,\ DeputyLevel \leftarrow 0,\ 0$;
3  $MainMatrix,\ DeputyMatrix \leftarrow Maintree,\ Deputytree$;
4  while $MainMatrix\ is\ not\ null$ do
5      $\xi_{s_i}^{cur}, \xi_{d_i}^{cur} = getPrefixCut(\xi_{s_i}, \xi_{d_i}, MainLevel)$;
6      for $k \leftarrow 0\ to\ p$ do
7          $bucketCur = MainMatrix.B(CandiAddresses(k))$;
8          if $\xi_{s_i}^{cur}, \xi_{d_i}^{cur},\ Indexpairs(k)\ match\ in\ bucketCur$ then
9              accumulate the weight and return true;
10     $nextLevel(\xi_{s_i}, \xi_{d_i}, MainMatrix, MainLevel)$;
11 while $DeputyMatrix\ is\ not\ null$ do
12     while $DeputyMatrix\ is\ virtual$ do
13         $nextLevel(\xi_{s_i}, \xi_{d_i}, DeputyMatrix, DeputyLevel)$;
14     $\xi_{s_i}^{cur}, \xi_{d_i}^{cur} = getPrefixCut(\xi_{s_i}, \xi_{d_i}, DeputyLevel)$;
15     for $k \leftarrow 0\ to\ p$ do
16         $bucketCur = MainMatrix.B(CandiAddresses(k))$;
17         if $\xi_{s_i}^{cur}, \xi_{d_i}^{cur},\ Indexpairs(k)\ match\ in\ bucketCur$ then
18             accumulate the weight and return true;
19         if $bucketCur\ is\ empty$ then
20             record $\xi$, index, w and return true;
21     if $DeputyLevel\ equals\ MainLevel + 1$ then
22         $cutAndRelink(MainLevel, DeputyLevel)$;
23         $Deputytree = generateMatrix(m, 0)$;
24         $DeputyMatrix \leftarrow Deputytree$;
25         $CurDeputyLevel \leftarrow 0$;
26     else
27         $Extend(DeputyLevel)$;
28         $nextLevel(\xi_{s_i}, \xi_{d_i}, DeputyMatrix, DeputyLevel)$;
```

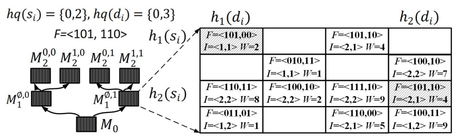


Figure 12: A running example of operations on the main tree

Extend. Algorithm 2 shows the operations of extend. When the matrix block *CurM* expands, Auxo first generates two new children matrix blocks, M_0 and M_1, with their children pointers initially set to null and the fingerprint length is one bit shortened. Then, Auxo sets the $child_0$ and $child_1$ pointers of *CurM* to the addresses of M_0 and M_1, respectively. Finally, Auxo moves the edges stored in matrix $CurM.B$ to matrices $M_0.B$ and $M_1.B$ according to their fingerprint prefixes, and releases the memory of $CurM.B$.

Edge Query. With a user-specified edge $x \rightarrow y$, the edge query returns its accumulated weight. On the Main tree, Auxo checks only one associated matrix on each level from the root to the leaf level by level according to the fingerprint pair of $x \rightarrow y$. Auxo checks all the candidate buckets in an associated matrix to match the index pair and the fingerprint pair with the prefix omitted. If Auxo finds a match item, it returns the stored weight value. If Auxo finds no match in the Main tree, it searches the edge in the Deputy tree. As the Deputy tree only has the leaf level of matrices, it only checks one associated matrix. If it still fails, Auxo returns a negative result.

Node Query. Given a user-specified node x, the node in(out)-flow query obtains the aggregated weight of all the in(out)-going edges of x. We only describe the operation of the node out-flow query as the node in-flow query involves similar operations. Algorithms 3 describe the detailed operation on the Main tree beginning at the root matrix (the same for the Deputy tree). In the following, we first present the operations of traversing all the matrices Auxo must check. Then, we focus on the operations in a single matrix. First, we take a closer look at each level of the Main tree to find the matrices which Auxo needs to check, while the operations on the Deputy tree are similar. Auxo starts with the root matrix M_0 on level 0. It is clear that M_0 should be checked. As the level 1 embeds the first bit of the destination node's fingerprint. This means an edge having source node x can be stored in $M_1^{\emptyset,0}$ or $M_1^{\emptyset,1}$. Therefore, the two matrices on level 1 should be checked (like line 17 in Algorithm 3). For level 2, Auxo embeds the first bit of the source node's fingerprint and it only needs to check $M_2^{0,0}$ and $M_2^{0,1}$, or $M_2^{1,0}$ and $M_2^{1,1}$ according to the first bit of ξ_x (like line 12 − 15 in Algorithm 3). Such a procedure continues until Auxo finishes searching the last level of the Main tree. According to the analysis of node query in Section 4.2, the number of matrices we have to check for node query is $O(\sqrt{n})$ where n indicates the total number of matrix blocks.

Algorithm 2: Extend($CurLevel$)

```
1  for $each\ CurM\ on\ CurLevel$ do
2      $M_0\ M_1 \leftarrow generateMatrix(m, CurLevel)$;
3      $set\ M_0.child_0, M_0.child_1, M_1.child_0, M_1.child_1\ equal\ null$;
4      $CurM.child_0, CurM.child_1\ \leftarrow M_0,\ M_1$;
5      $moveEdges(CurM, M_0, M_1, CurLevel)$;
6      $delete\ CurM.B$;
```

Algorithm 3: NORecurs($CurM,\ \xi_v,\ RowAddrs,\ Curlevel$)

```
1  $ResWeight \leftarrow 0$;
2  if $CurM$ is null then
3      return $ResWeight$;
4  if $CurM$ is valid then
5      for $i \leftarrow 0$ to $r$ do
6          for $j \leftarrow 0$ to $m$ do
7              $Curbucket = CurM.B(RowAddrs[i] \cdot m + j)$;
8              if $\xi_v$, $i$ match in $Curbucket$ then
9                  $ResWeight \leftarrow ResWeight + Curbucket.w$;
10 $Curlevel \leftarrow Curlevel + 1$;
11 if $Curlevel - 1$ is odd then
12     if $\xi_v$ starts with '0' then
13         return $ResWeight +$
               $NORecurs(CurM.child0,\ \xi_v, RowAddrs,\ Curlevel)$;
14     else
15         return $ResWeight +$
               $NORecurs(CurM.child1,\ \xi_v, RowAddrs,\ Curlevel)$;
16 else
17     return $ResWeight +$
           $NORecurs(CurM.child0,\ \xi_v, RowAddrs,\ Curlevel) +$
           $NORecurs(CurM.child1,\ \xi_v, RowAddrs,\ Curlevel)$;
```

Based on Figure 12, we give a simple example for node out-flow query. First, Auxo calculates the fingerprint $\xi_x = 101$ and the hash address sequence $hq(x) = \{0, 2\}$. Auxo first checks M_0. As level 1 embeds the first bit of the destination node's fingerprint, Auxo needs to check $M_1^{\varnothing,0}$ and $M_1^{\varnothing,1}$ on level 1. For level 2, since Auxo embeds the first bit of the source node's fingerprint, it only has to check $M_2^{1,0}$ and $M_2^{1,1}$ as $\xi_x = 110$ starts with "1", and so forth to the leaf level. In a single matrix, Auxo needs to check all the lines or columns of buckets corresponding to the hash address sequence (lines 6-13 in Algorithm 3). Consider the matrix $M_1^{\varnothing,1}$ on level one for example in Figure 12. As level one embed the first bit of the fingerprint of the destination node, for a node out-flow query, no prefix should be omitted. Then, according to the hash address sequence $hq(x) = \{0, 2\}$, we check the line 0 and 2 of the matrix $M_1^{\varnothing,1}$ with index "1" and "2", respectively. In bucket $M[0, 0]$, we find the fingerprint "101" and the index "1" both match those stored in the bucket. Thus, we should accumulate the weight value of two stored in the bucket. The same situation happens in bucket $M[2, 3]$. As the other buckets do not match the index and the fingerprint, we return six as the accumulated weight in $M_1^{\varnothing,1}$.

4 ANALYSIS

4.1 Accuracy Analysis

According to GSS [12], the weights of any two edges $s_1 \rightarrow d_1$ and $s_2 \rightarrow d_2$ are summarized if and only if $H(s_1) = H(s_2) \bigwedge H(d_1) = H(d_2)$, where $H(v) = (h(v) * 2^f) + \xi_v$. We can see that $H(v)$ has a range value of $range(H) = m \cdot 2^f$.

Theorem 4. *Auxo guarantees the relative error of the edge query as $P\{[\tilde{f}(s,d) - f(s,d)]/\overline{w} > \zeta\} \leq \frac{|E|}{\zeta \cdot range^2(H)}$, where $\tilde{f}(s,d)$ is the queried result; $f(s,d)$ is the ground truth; $\overline{w}$ is the average weight of edges, and $|E|$ is the size of the edge set in the graph stream.*

Proof. Considering a graph stream with $|E|$ distinct edges and accumulated weight $w_{(s,d)}$ for each edge $s \rightarrow d$. We use the variable X_{s_1,d_1} to denote the summarized weight stored in the bucket which records edge $s_1 \rightarrow d_1$. According to GSS [12], we obtain the expectation of X_{s_1,d_1} as below

$$E(X_{s_1,d_1}) \leq 1/range^2(H) \cdot \sum_{(s_2,d_2)\in E} w_{(s_2,d_2)} \tag{5}$$

Let $\beta = \sum w_{(s_2,d_2)}$, $(s_2, d_2) \in E$, according to the Markov inequality and Eq. (5), we obtain

$$\begin{aligned} P\{\tilde{f}(s,d) > f(s,d) + \eta\} &= P\{f(s,d) + X_{s,d} > f(s,d) + \eta\} \\ &\leq \frac{\beta}{\eta \cdot range^2(H)} \end{aligned} \tag{6}$$

We assume that the edges in the graph stream have an average weight of $\overline{w}$. Let $\zeta = \eta/\overline{w}$, we obtain $P\{[\tilde{f}(s,d) - f(s,d)]/\overline{w} > \zeta\} \leq \frac{|E|}{\zeta \cdot range^2(H)}$. Thus proved. The accuracy analysis of the node query is similar to that of the edge query accuracy. It is clear Auxo and GSS_Chain have the same accuracy if they have the same matrix width and fingerprint length (i.e., the same $range(H)$).

4.2 Time Cost Analysis

Insert and Edge Query. Theorem 2 shows the time cost for an Auxo to insert/query an edge is $O(log|E|)$. For a proportional Auxo, when inserting an edge, we may have to extend a full block. We show that this will not block the query operation and the amortized time complexity of insertion is still $O(log|E|)$.

Whenever extending a full block M, we generate two children blocks, i.e., M_0 and M_1, and free M after the edge moving operation finish. Thus, we can still query the result from M. For edge insertion, we assume that when inserting an edge, the average time of traversing a matrix is τ; when extending a block, the average time of moving an edge is ι ($\iota << \tau$), while a matrix contains an average number of n_a distinct edges. Considering an $l-$level proportional Auxo with $n = 2^l - 1$ blocks and a graph stream with no duplicated edges, we can decompose the time of constructing the l_{th} level into the time of the inserting edges IT_l and the time of moving edges MT_l. We have $IT_l \leq (n_a l\tau + n_a l\tau + 2n_a l\tau + ... + 2^{l-2} l n_a \tau) = 2^{l-1} l n_a \tau$, and $MT_l = (n_a\iota + 2n_a\iota + ... + 2^{l-2} n_a \iota) = (2^{l-1} - 1)n_a\iota$.

Thus, the amortized time of constructing the l_{th} level of a proportional Auxo is computed by Eq.(7),

$$\begin{aligned} AT_l &= (IT_l + MT_l)/2^{l-1}n_a \\ &\leq l\tau + (2^{l-1} - 1)\iota/2^{l-1} \end{aligned} \tag{7}$$

Therefore, the amortized time complexity of constructing the l_{th} level of a proportional Auxo is $O(l\tau) = O(logn) = O(log|E|)$.

Node Query. According to GSS [12], the time cost of the node query is $O(rm)$ in a single matrix, where r is the length of the hash address sequence, and m is the side width of the matrix. For GSS_Chain with n matrics, the time cost is $O(nrm)$ for the node query. We use the notations NO_i to quantify the number of matrices Auxo checks on level i for node out-flow query. On level 0, Auxo has only one matrix and $NO_0 = 1$. Considering the matrices at level i ($i > 0$), if Auxo embeds the next bit of the source node's fingerprint, we have $NO_i = NO_{i-1}$. Conversely, we have $NO_i =$

Table 4: Time consumption reduced by Auxo

levels	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$	$l=9$
Insert/edge query	0.73	0.84	0.90	0.94	0.97	0.98
Node out-flow query	0.40	0.58	0.67	0.77	0.82	0.91
Node in-flow query	0.60	0.68	0.78	0.83	0.88	0.94

$2NO_{i-1}$. Therefore, NO_i can be computed by Eq. (8).

$$NO_i = 2^{\lceil i/2 \rceil} \tag{8}$$

We use NO to denote the total number of matrices that should be checked for the node out-flow query. Thus, we have

$$NO = \sum_{i=0}^{l-1} NO_i = \begin{cases} 4 \cdot 2^k - 3, & l = 2k, \\ 6 \cdot 2^k - 3, & l = 2k+1. \end{cases} \quad k = 0, 1, 2, \ldots \tag{9}$$

Since an Auxo with $2k$ levels has $n = 2^{2k} - 1$ matrices, the time cost for node out-flow query is $O(rm\sqrt{n})$, which can also be computed by $O(rm\sqrt{\frac{|E|}{\alpha m^2}}) = O(r\sqrt{|E|})$. Due to symmetry, the node in-flow query has the same time cost of $O(r\sqrt{|E|})$. Eq. (8) shows the number of matrices Auxo needs to check on level l for the node query. When extending level l, Auxo extends level l at once, while proportional Auxo extends from level 0 to level l utilizing the Deputy tree. Therefore, the proportional Auxo always has less number of matrices to check on the last level of the Deputy tree and achieves better performance for node query.

Table 4 shows the time cost reduced by Auxo compared to GSS_Chain for different queries. Auxo reduces a large ratio of computation cost, especially for large-scale graph streams.

4.3 Memory Cost Analysis

It is clear that for an Auxo with l levels, the total memory cost is computed by Eq.(10),

$$\begin{aligned} M_A &= m^2 b(2^l - 1) - \sum_{i=0}^{l-1} m^2 i 2^i \\ &= m^2[b(2^l - 1) - 2^l(l-2) - 2] \end{aligned} \tag{10}$$

In Eq. (10), the variable b denotes the size of a bucket on level 0. Assuming the matrices have an average load factor of α, we obtain $l = log_2 \frac{|E|}{m^2\alpha}$ and we can get the space complexity $O(|E|(1-log|E|))$. In contrast, a GSS_Chain with $2^l - 1$ matrices consumes $m^2 b(2^l - 1)$ bits of memory. We can see that a full Auxo with l levels saves the memory cost by a ratio of $\frac{2^l(l-2)-2}{b(2^l-1)} \approx \frac{l-2}{b} = \frac{log_2 n - 2}{b} = [log_2|E| - log_2(4m^2\alpha)]/b$ of the space cost.

According to Eq. (4), a proportional Auxo with l levels on the Main tree has the worst bucket utilization of 0.75 when extending the $(l+1)_{th}$ level on the Deputy tree. The memory cost of the proportional Auxo is shown in Eq. (10), while the memory cost for GSS_Chain is at least $M_G = 0.75m^2 b(2^l - 1)$. Thus, we have $M_G - M_A \approx m^2 2^l(l - 2 - 0.25b)$. When $l > 0.25b + 2$ ($|E| > m^2\alpha 2^{(0.25b+2)}$, equally), the proportional Auxo consumes less memory than GSS_Chain, even in the worst case.

4.4 Load Factor Analysis

As we have aforementioned, in each matrix, Auxo allocates p candidate buckets for an edge to improve the bucket utilization of the matrix. A high load factor is important before we scale Auxo. Considering a single matrix, we use q_i to denote the probability that the i_{th} edge can be inserted successfully. We can obtain q_i as

$$q_i = q_{i-1}[1 - (\frac{i-1}{m^2})^p], \; i = 1, 2, ..., m^2 + 1; \; q_0 = 1 \tag{11}$$

The matrix has a load factor of $(i-1)/m^2$ if and only if the $(i-1)_{th}$ edge is inserted successfully, while the insertion of the i_{th} edges can fail with the probability of $q_{i-1} - q_i$. Therefore, the expectation of load factor $E(\alpha)$ is computed by Eq.(12),

$$E(\alpha) = \sum_{i=1}^{m^2} \frac{i}{m^2}[q_i - q_{i+1}] \tag{12}$$

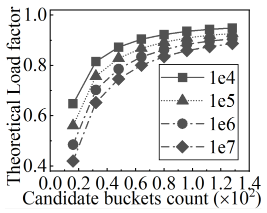


Figure 13: The expectation of load factor

Combining Eqs. (11) and (12), we can achieve $E(\alpha)$. Figure 13 shows the theoretical $E(\alpha)$ with different matrix sizes and numbers of candidate buckets when we fix the length of the hash address sequence at $r = 16$. For all the matrix sizes, when the numbers of the candidate bucket p increase from 16 to 80, the load factors increase to 80% and all reach nearly 90% when $p = 128$.

5 PERFORMANCE EVALUATION

5.1 Experiment Setups

We have implemented Auxo and made the source code publicly available. We also implement the baseline scheme GSS_Chain over the open-sourced code of the state-of-the-art GSS design [12]. We obtain the source codes of other related work including Scube [7], Horae [8], MoSSo [18], SGS [22], and GS4 [2], with which we compare the performance of Auxo with those of all of the baseline schemes. We conduct all the experiments on a machine with a 16-core 2.4GHz Xeon CPU, 64GB RAM, and 1TB HDD.

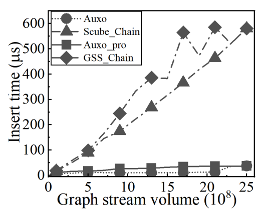
In the experiments, we use three types of datasets including hyperlink network, social network, and IP network. To guarantee the fairness of comparison and well evaluate the scaling efficiency of the structures, we collect five large-scale graph stream datasets from real world systems which cover three categories.

1) **Friendster** [19]: This is the friendship network of the online social site Friendster, where nodes represent users and a directed edge denotes that a user adds another user to his or her friend list. The dataset contains 68 million nodes and 2.6 billion edges.

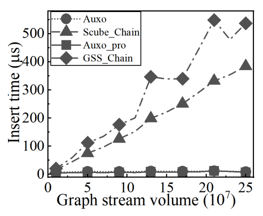
2) **UK-2002** [19]: This is the hyperlink network of the UK domain for the United Kingdom in 2002 with 18.5 million nodes and 262 million edges.

3) **Delicious-ui** [19]: It is a bipartite network where source nodes represent users while destination nodes represent URLs. An edge means a user tagged an URL. This network has 34.6 million nodes and 301 million edges.

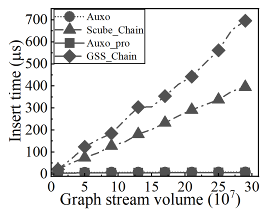
4) **Caida** [5]: This dataset contains anonymous passive traffic traces on high-speed Internet backbone links in Chicago in February 2015 for ten minutes. Nodes represent IP addresses, and a weighted



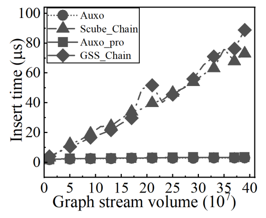
(a) Friendster



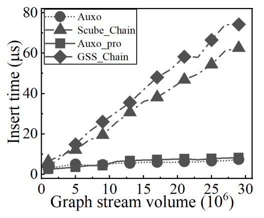
(b) UK-2002



(c) Delicious-ui

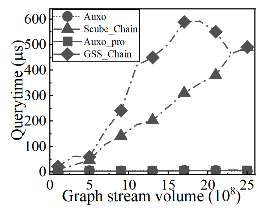


(d) Caida

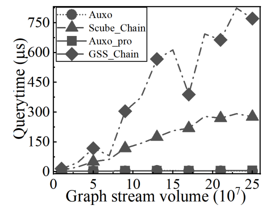


(e) DBLP

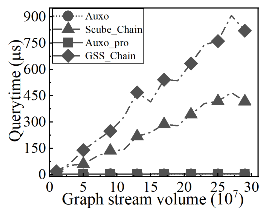
Figure 14: The per edge insertion time with different graph stream volume



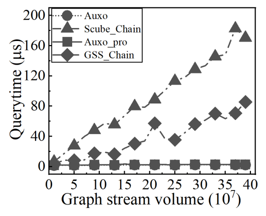
(a) Friendster



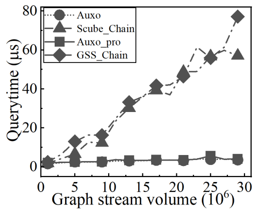
(b) UK-2002



(c) Delicious-ui



(d) Caida



(e) DBLP

Figure 15: The edge query time with different graph stream volume

edge measures the size of the data packet. The dataset contains 2.1 million nodes and 403 million edges.

5) **DBLP** [19]: This is a network of collaboration collected from the DBLP computer science bibliography where nodes represent researchers, and edges represent paper co-authorship. The dataset includes 16.7 million nodes and 30 million edges.

For parameter setting of GSS_Chain and Auxo, according to Theorem 4, we set the fingerprint length to guarantee the edge query accuracy satisfying $P\{[\tilde{f}(s,d) - f(s,d)]/\overline{w} > 0.01\} \leq 0.01$. Then, we choose the length of the hash address sequence r and the number of candidate buckets p by Eq. (12). For other structures, we follow the recommended parameter settings. We vary the volume of the inserted edges and compare their performance. We use the Blizzard hash algorithm and generate a 64-bit hash value $hash(v)$. The fingerprint ξ_v and the origin hash address $h(v)$ are computed as $\xi_v = hash(v)\%2^f,\ h(v) = \lfloor \frac{hash(v)}{2^f} \rfloor \%m$.

5.2 Metrics

Time-Accumulated Allocated Memory (TAM) measures the overall memory overhead for a graph stream summarization structure that scales with time. For a time range $[T_0, T_1]$, we use $AM(t)$ to denote the allocated memory for the structure at time instance t. We define $TAM = \int_{T_0}^{T_1} AM(t)\ dt$ as the integral of the allocated memory over time. In cloud services, service providers typically charge based on the amount of resources allocated and the length of the service time. Therefore, TAM directly reflects the overall memory overhead of a scalable graph stream summarization.

Average Relative Error (ARE) measures the accuracy of aggregation queries (e.g., edge/node query). It is clear for these queries, the results can be over-estimated. For a set of aggregation queries $Q = \{q_1, q_2, ..., q_n\}$ sized n, we let $f(q_i)$ and $\tilde{f}(q_i)$ represent the ground truth and the queried result, respectively. Then, we define the relative error $RE(q_i)$ of q_i as $RE(q_i) = \frac{\tilde{f}(q_i) - f(q_i)}{f(q_i)}$ and the average relative error $ARE(Q)$ of query set Q as $ARE(Q) = \frac{\sum_{i=1}^{n} RE(q_i)}{n}$.

According to the compressed matrix structure of GSS [12], with edge/node query, we can build a sketched graph stream with the topology structure preserved. Therefore, Auxo can support more graph topology-related queries. In the experiment, we evaluate Auxo with the reachability query [38] and triangle counting [4, 9].

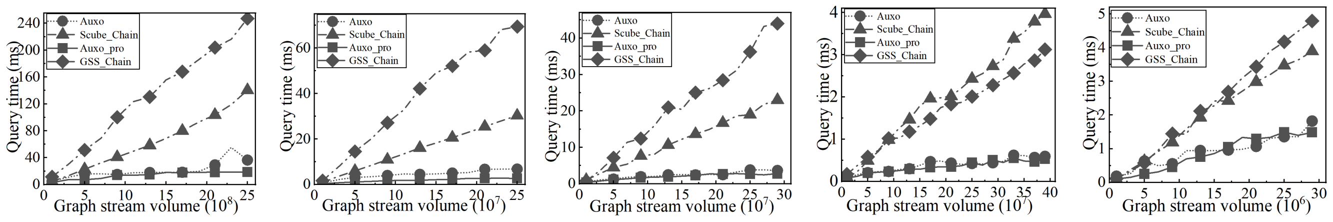
True Negative Recall measures the precision of the reachability query. Given a node pair (s, d), the reachability query returns whether there is a path from s to d. It is clear that GSS_Chain and Auxo have no false negatives (i.e., if d is reachable from s in the graph stream, Auxo definitely returns true; otherwise, it may return true with a small possibility). For a set of reachability queries and all the node pairs are unreachable. The true negative recall is defined as the ratio of node pairs reported as unreachable.

5.3 Results

We conduct six groups of experiments to examine our design. First, we evaluate the execution time of different operations and the memory cost of Auxo, GSS_Chain, and Scube [7]. Second, we compare Auxo with other scalable graph stream summarization structures like Horae [8], MoSSo [18], SGS [22], and GS4 [2]. Third, we test the accuracy for different queries as well as the load factor of the matrices. Fourth, we evaluate two compound graph algorithms on the graph stream summarization structures. Fifth, we examine parallel optimizations for Auxo to further accelerate the speed of operations. Finally, we utilize a large-scale real-world graph stream as a case study to evaluate the efficiency and effectiveness of Auxo. For simplicity, in the following, we use “Auxo_pro” to represent “the proportional Auxo” for short.

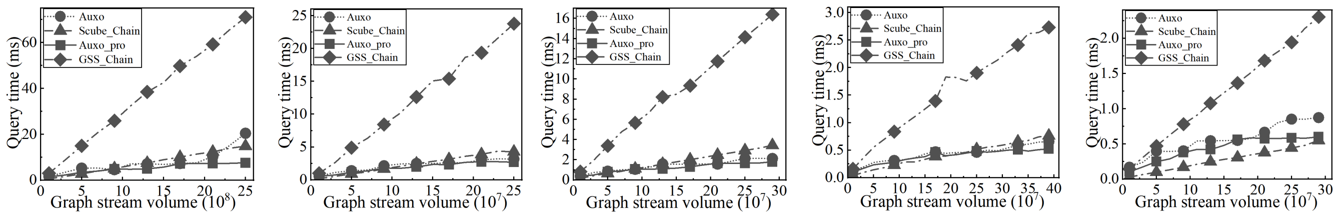
When an insertion fails, Scube [7] does not scale but allocates more candidate buckets for an edge. Hence, we follow the chaining style to scale the Scube structure. Once the load factor of the current compressed matrix reaches 85%, we append another Scube block (a compressed matrix with a degree estimator) and insert the edges into the newly generated one. We call such a structure Scube_Chain.

Edge Insertion. Figure 14 shows that both Auxo and Auxo_pro reduce the edge insertion time of GSS_Chain and Scube_Chain by



(a) Friendster (b) UK-2002 (c) Delicious-ui (d) Caida (e) DBLP

Figure 16: The node in-flow query time with different graph stream volume



(a) Friendster (b) UK-2002 (c) Delicious-ui (d) Caida (e) DBLP

Figure 17: The node out-flow query time with different graph stream volume

one to two orders of magnitude over various scales of datasets. We can get the same conclusion from the overall throughput reported in Figure 18.

Edge Query. Figure 15 shows that both Auxo and Auxo_pro reduce the edge query time by one to two orders of magnitude for all the data sets compared with GSS_Chain and Scube_Chain.

Node Query. Figures 16 and 17 show the query time of node query. For node in-flow query, both Auxo and Auxo_pro outperform GSS_Chain and Scube_Chain, reducing the query time by about one order of magnitude for different datasets. For node out-flow query, Auxo and Scube_Chain achieve comparable performance, both outperformance GSS_Chain. To further evaluate the time cost of Auxo and Auxo_pro on node query, we examine the node query latency whenever Auxo has just extended a new level on Delicious-ui. Figure 19 shows that Auxo_pro reduces the node in(out)-flow query latency of Auxo by at most 37% and 25% respectively.

Memory Cost and Memory Utilization. Table 5 shows the total memory cost. Auxo_pro reduces 14% of the memory cost on average compared to GSS_Chain while Scube costs much more memory. It is worth noting that the Caida dataset has a large proportion of duplicated edges. As for duplicate appeared edges, we just accumulate the weights. Therefore, it cost much less memory than expected. Figure 22 shows the normalized TAM and Auxo_pro reduces the TAM of Auxo by a ratio of 34% on average. In cloud service, it means a 34% reduction of cost on memory. Figure 23 shows that Auxo_pro has a much higher average memory utilization (over 80%) compared to the basic Auxo (about 60%).

Comparison with Other Scalable Structures. We compare Auxo with other scalable graph stream summarization structures including Horae [8], MoSSo [18], and SGS [22]. In the experiment, we mainly examine the update speed and space cost. Since Horae scales as time stamp increasing, we set the time unit as one million tuples. We set all the parameters as recommended. Figures 20 and 21 show the results on the UK-2002 dataset. Horae and Auxo achieve nearly the same performance on update speed and are about two orders of magnitude faster than MoSSo and SGS. Moreover, Auxo

Table 5: Total memory cost (G)

Data sets	Proportional Auxo	GSS_Chain	Scube_Chain
Friendster	13.07	15.68	29.11
UK-2002	1.52	1.7	2.9
Delicious-ui	0.51	0.62	1.27
Caida	0.045	0.051	0.153
DBLP	0.091	0.104	0.213

reduces the memory cost by more than one order of magnitude compared to other structures. We do not show the result of GS4 [2] here as it costs nearly 30 hours to summarize a graph stream with an edge count of only 700, 000 which makes it not comparable.

Accuracy and Load Factor. As aforementioned in Section 4.1, three data structures (GSS_Chain, Auxo, proportional Auxo) have the same accuracy when we set the matrix's side width and fingerprint length to the same. Here, we set the fingerprint length to 16, 20, and 24, respectively. Figures 24 and 25 show the ARE of different queries as the graph stream's volume scale. The results show that we can guarantee accuracy by adjusting the length of the fingerprint. We further examine the average load factor of all the full matrices to compare with the theoretical analysis. We set the initial matrix size to 100, 000 with $r = 16$ and vary the number of candidate buckets. Figure 26 shows the average load factors of the three data structures. We can see that for all three data structures, the experimental results are pretty close to the theoretical analysis.

Graph Queries. For the reachability query, we take 200 unreachable node pairs and run the BFS search to query the reachability. Figure 27 shows that the true negative recall on dataset Caida is pretty high, nearly 100% for different fingerprint lengths. Figure 28 shows the time cost for the reachability query. Auxo and Auxo_pro both outperform GSS_Chain and Scube_Chain by nearly one order of magnitude. In a graph, a triangle is a triple of three edges where every two edges share a common node. We examine the accuracy of Auxo on triangle counting query. Figure 29 shows that the relative error of the triangle counting query on dataset DBLP is pretty low.

Parallel Optimization. It is clear that for Auxo, the operations in different matrices are independent, and thus it can parallelize

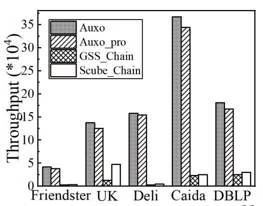


Figure 18: Overall throughput (tuple/s)

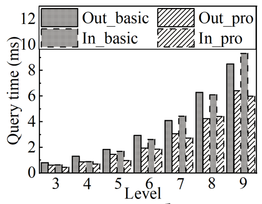


Figure 19: Node query time for various levels

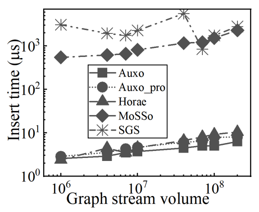


Figure 20: Structures' edge updating time

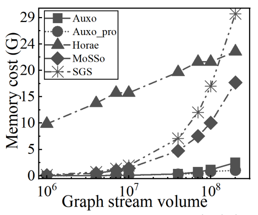


Figure 21: scalable structures' space cost

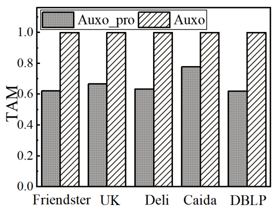


Figure 22: The normalized TAM

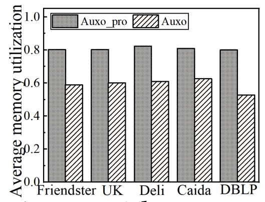


Figure 23: The average memory utilization

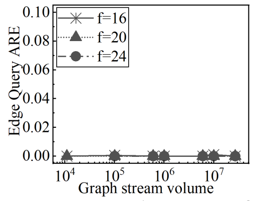


Figure 24: The ARE of edge query for DBLP

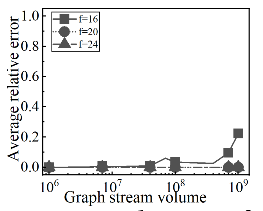


Figure 25: The ARE of node in-flow query

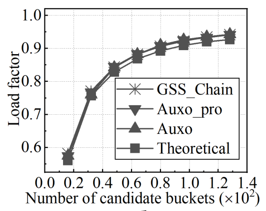


Figure 26: The average matrices load factor

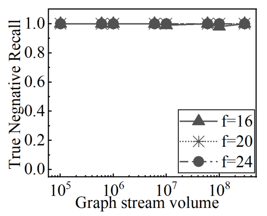


Figure 27: True negative recall

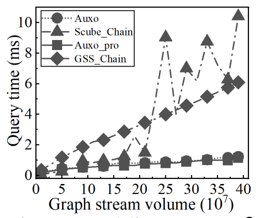


Figure 28: Time cost of reachability query

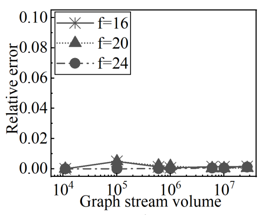


Figure 29: Relative error of triangle count

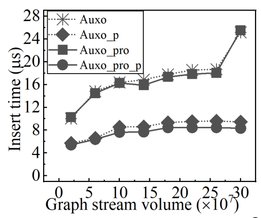


Figure 30: Time cost of the parallelized insert

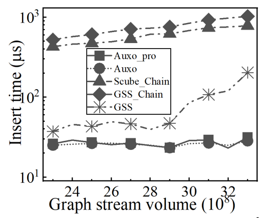


Figure 31: Structures' updating time

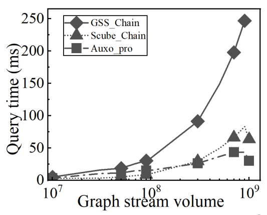


Figure 32: Time cost of node similarity query

the operation without any locks for edge query and node query, etc. We parallelize the operations on Auxo and proportional Auxo and take the insert query on the dataset Delicious-ui as examples. Figures 30 show that Auxo further achieves about 3× speedup on insert query with eight paralleling threads.

Case Study. We collect a large-scale hyperlink network graph stream dataset of the United Kingdom in 2007. Unlike the dataset UK-2002, UK-2007 [3] is a much large dataset containing 105 million nodes and 3.3 billion edges. Figure 31 shows the insertion time for different structures. We can see that both Auxo and the proportional Auxo outperform GSS_Chain and Scube_Chain for one to two orders of magnitude. We also examine the performance of GSS (without chaining and inserting the edge into the buffer if the compressed matrix is full) and it costs nearly an order of magnitude more time to insert an edge compared with Auxo in the case of large-scale graph stream. Table 6 shows the memory cost. GSS costs about six times greater memory than our design. Such a poor performance of GSS is caused by its large buffer.

We also design a node-neighbor similarity experiment (which is a critical metric in web-page clustering) to test the effectiveness of Auxo. We use the Jaccard coefficient $J(N_x, N_y) = |N_x \cap N_y|/|N_x \cup N_y|$ to measure the similarity of two sets, where N_v indicates the neighbor set of node v. Here we consider node y to be node x's neighbor if and only if there exists an edge $x \longrightarrow y$ or $y \longrightarrow x$. Figure 32 shows Auxo takes less than 40 ms to conduct such a query on a graph with a volume over 10^9.

Table 6: Memory cost of UK-2007 (G)

Proportional Auxo	GSS_Chain	Scube_Chain	GSS
18.22	20.05	32.73	103.86

6 CONCLUSION AND FUTURE WORK

In this paper, we propose Auxo, a scalable and efficient structure for graph stream summarization. Auxo designs a *prefix embedded tree* (PET) framework and achieves good time/memory efficiency. Auxo reduces the query time to $log|E|$ scale as well as saves the space cost by $\frac{log_2 n-2}{b}$. Furthermore, we propose a novel proportional incremental strategy to improve memory utilization. Comprehensive experiments on large-scale datasets show that Auxo reduces the time costs of insertion, edge query, and node query by one to two orders of magnitude compared to state-of-the-art designs. Moreover, Auxo achieves efficiently and economically structure scaling with an average memory utilization of over 80%. In the next step, we plan to explore two issues of a scalable graph stream summarization structure, including accuracy guarantee as graph stream volume increases continuously and the overflow of the counters for the accumulated weights, especially in the presence of highly duplicated edges in a graph stream.

ACKNOWLEDGMENTS

This research is supported in part by NSFC under grant No. 61972446 and Huawei Research Project No. TC20210702017.

REFERENCES

[1] Charu C. Aggarwal, Yao Li, and Philip S. Yu. 2020. On Supervised Change Detection in Graph Streams. In *Proceedings of the 2020 SIAM International Conference on Data Mining (SDM '20)*. SIAM, Ohio, USA, May 7-9, 2020, 7–9. https://doi.org/10.1137/1.9781611976236.33

[2] Nosratali Ashrafi-Payaman, Mohammadreza Kangavari, Saeid Hosseini, and Amir Mohammad Fander. 2021. GS4: Graph stream summarization based on both the structure and semantics. *J. Supercomput.* 77, 3 (2021), 2713–2733. https://doi.org/10.1007/s11227-020-03290-2

[3] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: a scalable fully distributed Web crawler. *Softw. Pract. Exp.* 34, 8 (2004), 711–726. https://doi.org/10.1002/spe.587

[4] Vladimir Braverman, Rafail Ostrovsky, and Dan Vilenchik. 2013. How Hard Is Counting Triangles in the Streaming Model?. In *Proceedings of the 40th International Colloquium on Automata, Languages and Programming (ICALP '13)*. Springer, Riga, Latvia, July 8-12, 2013, 244–254. https://doi.org/10.1007/978-3-642-39206-1_21

[5] CAIDA. 2022. *Anonymized Internet Traces 2015*. The University of California. Retrieved 28 April, 2022 from https://catalog.caida.org/details/dataset/passive_2015_pcap

[6] Hanhua Chen, Hai Jin, and Shaoliang Wu. 2016. Minimizing Inter-Server Communications by Exploiting Self-Similarity in Online Social Networks. *IEEE Trans. Parallel Distributed Syst.* 27, 4 (2016), 1116–1130. https://doi.org/10.1109/TPDS.2015.2427155

[7] Ming Chen, Renxiang Zhou, Hanhua Chen, and Hai Jin. 2022. Scube: Efficient Summarization for Skewed Graph Streams. In *Proceedings of the 42nd IEEE International Conference on Distributed Computing Systems (ICDCS '22)*. IEEE, Bologna, Italy, July 10-13, 2022, 100–110. https://doi.org/10.1109/ICDCS54860.2022.00019

[8] Ming Chen, Renxiang Zhou, Hanhua Chen, Jiang Xiao, Hai Jin, and Bo Li. 2022. Horae: A Graph Stream Summarization Structure for Efficient Temporal Range Query. In *Proceedings of the 38th IEEE International Conference on Data Engineering (ICDE '22)*. IEEE, Kuala Lumpur, Malaysia, May 9-12, 2022, 2792–2804. https://doi.org/10.1109/ICDE53745.2022.00254

[9] Michael Elkin. 2011. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Trans. Algorithms* 7, 2 (2011), 20:1–20:17. https://doi.org/10.1145/1921659.1921666

[10] Neha Ghoshon. 2020. *COVID-19: What Is Asymptomatic Transmission? This Is How You Could Be Spreading It!* Greynium Information Technologies Pvt. Ltd. Retrieved 7 April, 2022 from https://www.boldsky.com/health/wellness/asymptomatic-transmission-of-covid-19-132748.html

[11] Xiangyang Gou and Lei Zou. 2021. Sliding Window-based Approximate Triangle Counting over Streaming Graphs with Duplicate Edges. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. ACM, Xi'an, Shaanxi, China, June 20-25, 2021, 645–657. https://doi.org/10.1145/3448016.3452800

[12] Xiangyang Gou, Lei Zou, Chenxingyu Zhao, and Tong Yang. 2019. Fast and Accurate Graph Stream Summarization. In *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE '19)*. IEEE, Macao, China, April 8-11, 2019, 1118–1129. https://doi.org/10.1109/ICDE.2019.00103

[13] Huawei GTS. 2023. *Telecom-graph*. DataFountain. Retrieved 12 February, 2023 from https://www.datafountain.cn/datasets/6847

[14] Sudipto Guha and Andrew McGregor. 2012. Graph Synopses, Sketches, and Streams: A Survey. *Proc. VLDB Endow.* 5, 12 (2012), 2030–2031. https://doi.org/10.14778/2367502.2367570

[15] Abdul Hameed, Alireza Khoshkbarforoushha, Rajiv Ranjan, Prem Prakash Jayaraman, Joanna Kolodziej, Pavan Balaji, Sherali Zeadally, Qutaibah Marwan Malluhi, Nikos Tziritas, Abhinav Vishnu, Samee U. Khan, and Albert Y. Zomaya. 2016. A survey and taxonomy on energy efficient resource allocation techniques for cloud computing systems. *Computing* 98, 7 (2016), 751–774. https://doi.org/10.1007/s00607-014-0407-8

[16] Mohamed S. Hassan, Bruno Ribeiro, and Walid G. Aref. 2018. SBG-sketch: a self-balanced sketch for labeled-graph stream summarization. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management (SSDBM '18)*. ACM, Bozen-Bolzano, Italy, July 09-11, 2018, 3:1–3:12. https://doi.org/10.1145/3221269.3223030

[17] Changsheng Hou, Bingnan Hou, Tongqing Zhou, and Zhiping Cai. 2021. DMatrix: Toward fast and accurate queries in graph stream. *Comput. Networks* 198 (2021), 108403. https://doi.org/10.1016/j.comnet.2021.108403

[18] Jihoon Ko, Yunbum Kook, and Kijung Shin. 2020. Incremental Lossless Graph Summarization. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD '20)*. ACM, Virtual Event, CA, USA, August 23-27, 2020, 317–327. https://doi.org/10.1145/3394486.3403074

[19] Jerome Kunegis. 2013. KONECT: the Koblenz network collection. In *Proceedings of the 22nd International World Wide Web Conference (WWW '13)*. ACM, Rio de Janeiro, Brazil, May 13-17, 2013, 1343–1350. https://doi.org/10.1145/2487788.2488173

[20] Pierre L'Ecuyer. 1999. Tables of linear congruential generators of different sizes and good lattice structure. *Math. Comput.* 68, 225 (1999), 249–260. https://doi.org/10.1090/S0025-5718-99-00996-5

[21] Youhuan Li, Lei Zou, M. Tamer Özsu, and Dongyan Zhao. 2019. Time Constrained Continuous Subgraph Search Over Streaming Graphs. In *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE '19)*. IEEE, Macao, China, April 8-11, 2019, 1082–1093. https://doi.org/10.1109/ICDE.2019.00100

[22] Ziyi Ma, Jianye Yang, Kenli Li, Yuling Liu, Xu Zhou, and Yikun Hu. 2021. A Parameter-Free Approach for Lossless Streaming Graph Summarization. In *Proceedings of the 26th International Conference on Database Systems for Advanced Applications (DASFAA '21)*. Springer, Taipei, Taiwan, April 11-14, 2021, 385–393. https://doi.org/10.1007/978-3-030-73194-6_26

[23] Roger Magoulas and Steve Swoyer. 2020. *Cloud Adoption in 2020*. O'Reilly Media, Inc. Retrieved May 19, 2022 from https://www.oreilly.com/radar/cloud-adoption-in-2020/

[24] Sunilkumar S. Manvi and Gopal Krishna Shyam. 2014. Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey. *J. Netw. Comput. Appl.* 41 (2014), 424–440. https://doi.org/10.1016/j.jnca.2013.10.004

[25] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2020. Regular Path Query Evaluation on Streaming Graphs. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD '20)*. ACM, online conference, June 14-19, 2020, 1415–1430. https://doi.org/10.1145/3318464.3389733

[26] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2022. Evaluating Complex Queries on Streaming Graphs. In *Proceedings of the 38th IEEE International Conference on Data Engineering (ICDE '22)*. IEEE, Kuala Lumpur, Malaysia, May 9-12, 2022, 272–285. https://doi.org/10.1109/ICDE53745.2022.00025

[27] Ramesh Paudel and William Eberle. 2020. An Approach For Concept Drift Detection in a Graph Stream Using Discriminative Subgraphs. *ACM Trans. Knowl. Discov. Data* 14, 6 (2020), 70:1–70:25. https://doi.org/10.1145/3406243

[28] Jorma Rissanen. 1978. Modeling by shortest data description. *Autom.* 14, 5 (1978), 465–471. https://doi.org/10.1016/0005-1098(78)90005-5

[29] Neil Shah, Danai Koutra, Tianmin Zou, Brian Gallagher, and Christos Faloutsos. 2015. TimeCrunch: Interpretable Dynamic Graph Summarization. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining5 (SIGKDD '15)*. ACM, Sydney, NSW, August 10-13, 2015, 1055–1064. https://doi.org/10.1145/2783258.2783321

[30] Aida Sheshbolouki and M. Tamer Özsu. 2022. sGrapp: Butterfly Approximation in Streaming Graphs. *ACM Trans. Knowl. Discov. Data* 16, 4 (2022), 76:1–76:43. https://doi.org/10.1145/3495011

[31] Sukhpal Singh and Inderveer Chana. 2016. Cloud resource provisioning: survey, status and future research directions. *Knowl. Inf. Syst.* 49, 3 (2016), 1005–1069. https://doi.org/10.1007/s10115-016-0922-3

[32] Chunyao Song, Tingjian Ge, Yao Ge, Haowen Zhang, and Xiaojie Yuan. 2019. Labeled graph sketches: Keeping up with real-time graph streams. *Inf. Sci.* 503 (2019), 469–492. https://doi.org/10.1016/j.ins.2019.07.019

[33] Lorenzo De Stefani, Erisa Terolli, and Eli Upfal. 2021. Tiered Sampling: An Efficient Method for Counting Sparse Motifs in Massive Graph Streams. *ACM Trans. Knowl. Discov. Data* 15, 5 (2021), 79:1–79:52. https://doi.org/10.1145/3441299

[34] Nan Tang, Qing Chen, and Prasenjit Mitra. 2016. Graph Stream Summarization: From Big Bang to Big Crunch. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, Francisco, CA, USA, June 26-July 01, 2016, 1481–1496. https://doi.org/10.1145/2882903.2915223

[35] Tencent. 2021. *Tencent Health Code statistics. (2021)*. Tencent. Retrieved November 15, 2022 from https://www.tencent.com/zh-cn/business/health-code.html

[36] Ioanna Tsalouchidou, Francesco Bonchi, Gianmarco De Francisci Morales, and Ricardo Baeza-Yates. 2020. Scalable Dynamic Graph Summarization. *IEEE Trans. Knowl. Data Eng.* 32, 2 (2020), 360–373. https://doi.org/10.1109/TKDE.2018.2884471

[37] Tangwei Ying, Hanhua Chen, and Hai Jin. 2020. Pensieve: Skewness-Aware Version Switching for Efficient Graph Processing. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD '20)*. ACM, online conference, June 14-19, 2020, 699–713. https://doi.org/10.1145/3318464.3380590

[38] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability queries on large dynamic graphs: a total order approach. In *Proceedings of the 2014 International Conference on Management of Data (SIGMOD '14)*. ACM, Snowbird, UT, USA, June 22-27, 2014, 1323–1334. https://doi.org/10.1145/2588555.2612181