Scube: Efficient Summarization for Skewed Graph Streams

Ming Chen, Renxiang Zhou, Hanhua Chen, Hai Jin
National Engineering Research Center for Big Data Technology and System
Cluster and Grid Computing Lab
Services Computing Technology and System Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, China
{mingc, mr_zhou, chen, hjin}@hust.edu.cn

***Abstract*—Graph stream, which represents an evolving graph updating as an infinite edge stream, is a special emerging graph data model widely adopted in big data analysis applications. Entirely storing the continuously produced and tremendously large-scale datasets is impractical. Therefore, graph stream summarization structures which support approximate graph stream storage and management attract much recent attention. Existing designs commonly leverage a compressive matrix and use hash-based schemes to map each edge to a bucket of the matrix. Accordingly, they store the edges associated with the same node in the same row or column of the matrix. We show that existing designs suffer from unacceptable query latency and precision in the presence of node degree skewness in graph streams.**

We argue that the key to efficient graph stream summarization is to identify the high-degree nodes and leverage a differentiated strategy for the associated edges. However, it is not trivial to estimate the degree of a node in real-time graph streams due to the rigorous requirements of space and time efficiency. Moreover, the existence of duplicate edges makes high-degree nodes identification difficult. To solve the problem, we propose Scube, an efficient summarization structure for skewed graph streams. Two factors contribute to the efficiency of Scube. First, Scube proposes a space and computation efficient probabilistic counting scheme to identify high-degree nodes in a graph stream. Second, Scube differentiates the storage strategy for the edges associated with high-degree nodes by dynamically allocating multiple rows or columns. We conduct comprehensive experiments to evaluate the performance of Scube on large-scale real-world datasets. The results show that Scube significantly reduces the query latency over a graph stream by 48%-99%, as well as achieving acceptable query accuracy compared to the state-of-the-art designs.

***Index Terms*—Graph stream, summarization, skewness**

I. Introduction

Graph stream, a recent emerging graph data model, represents a fast evolving graph with an infinite stream of elements [1], [2]. Each element in the graph stream is represented as (s_i, d_i, w_i, t_i) $(i > 0)$, denoting a directed edge $s_i \rightarrow d_i$ with a weight value w_i generated at time t_i. Such a graph data model is widely used in big data applications, such as cybersecurity and surveillance [3], user behavior analysis in e-commerce networks [1], close contact tracking in the epidemic prevention and control [4], and user interactions analysis in social networks [5].

*The corresponding author is Hanhua Chen (chen@hust.edu.cn).

Emerging big data application systems produce tremendously large-scale and real-time graph stream data. For example, in WeChat, 902 million active users share 38 billion messages every day [6]. Tencent's Health Code platform has more than one billion users who have generated more than 24 billion health code scans during the anti-epidemic campaign of COVID-19 [7]. To cope with such large-scale infinite datasets, graph stream summarization becomes a practical technique that attracts lots of recent efforts [8]–[10].

TCM [8] is the first graph stream summarization structure, which is basically an $m \times m$ matrix M with all the buckets initially set to zero. It uses a hash function $h(\cdot)$ with the value range [0, m) to map each element (s_i, d_i, w_i, t_i) to the $h(s_i)^{th}$ row and the $h(d_i)^{th}$ column of the matrix by adding w_i to the value of the corresponding bucket. Given a specified edge, TCM can check whether or not it has appeared. It can also support more complex queries, e.g., obtaining the aggregated weight value of a specified edge and returning the aggregated weight of all the incoming edges of a specified node. However, the hash-based scheme can lead to poor query precision due to hash collisions. To improve the accuracy of query results, Gou *et al.* [9] propose GSS which stores a pair of fingerprints associated with an edge together with its weight value in the corresponding bucket. If a later coming edge with different fingerprints is mapped into the same bucket, GSS inserts the edge information into an extra buffer. A large-scale buffer can lead to a long query latency.

With hash mapping, existing graph stream summarization structures map the edges associated with the same node into the same row or column of the compressive matrix. Hence, in the precence of skewed graph streams [11], they can suffer from poor query precision and long query latency. Figure 1 plots the node degree distributions of different real-world datasets, which reveal high skewness, *i.e.*, the number of the distinct edges associated with a high-degree node can be very large. With existing designs, all the edges associated with the same high-degree node are mapped to the same row or column, resulting in severe hash collisions. Figure 2 examines the *average relative error* (ARE) of the edge aggregation weight query over TCM [8] with the dbpedia dataset [12] (The relative error is computed by $(\widetilde{Q} - Q)/Q$, where Q represents

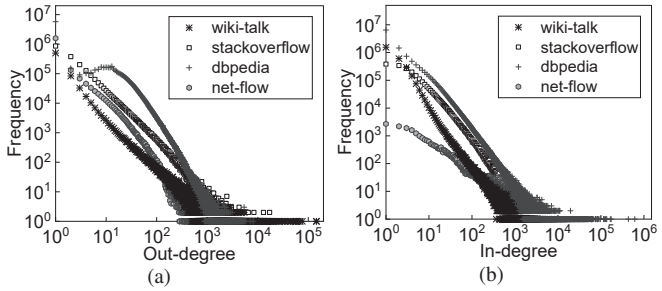

Fig. 1. Node degree distribution in real-world graph streams

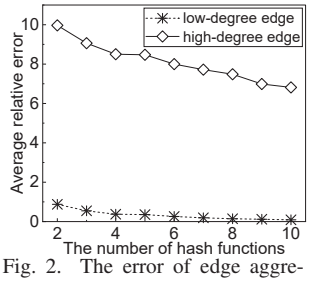

Fig. 2. The error of edge aggregation weight queries in TCM on dataset dbpedia

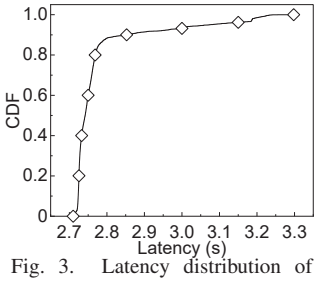

Fig. 3. Latency distribution of node aggregation weight queries in GSS on dataset dbpedia

the exact value and $\widetilde{Q}$ represents the query result. The ARE averages all the relative error values in a query set). The result in Fig. 2 shows unacceptable accuracy of the queries about the edges with high-degree nodes. Figure 3 plots the distribution of the latency of node aggregation weight queries over GSS [9], which shows prohibitively long query latency because of the hash collisions raised by high-degree nodes.

Based on the above observations, we argue that the key to efficient graph stream summarization is to identify high-degree nodes and use a differentiated storage strategy for their associated edges. However, identifying high-degree nodes in a graph stream is difficult. First, a graph stream evolves fast and the dataset can become tremendously large, raising rigorous requirements of both computation and memory efficiency in designing a node degree estimation scheme. Second, edge duplicating is common in graph streams and this makes accurately node degree estimating a challenging issue.

In this work, we propose Scube, a novel summarization structure for skewed graph streams. Two factors contribute to the efficiency of Scube. First, Scube designs a time and space probabilistic counting scheme to identify the high-degree nodes in graph streams. The proposed scheme only relies on light weight computation on a simple bit vector. Second, Scube proposes a differentiated strategy for storing the edges associated with high-degree nodes, which uses a dynamic address allocation method to assign more space for high-degree nodes and successfully alleviate hash collisions.

We implement Scube and conduct comprehensive experiments to evaluate the performance on large-scale real-world datasets. The results show that Scube significantly reduces the query processing latency by 48%-99% compared to the state-of-the-art designs while achieving acceptable accuracy.

The rest of this paper is organized as follows. Section II reviews the related work. Section III presents the design of Scube. Section IV describes the operations of Scube in detail. Section V analyzes the overhead and the collision rate of Scube. Section VI evaluates the performance of Scube. Section VII concludes the paper.

II. Related Work

To cope with the emerging large-scale graph streams [1], [2], [8], [9], [13], graph stream summarization techniques attract lots of recent research efforts [8], [9], [14].

Tang *et al.* propose TCM [8], which uses an $m \times m$ hash-based compressive matrix M to approximately represent a graph stream. Initially, the values of all the buckets of the matrix are set to zero. For inserting an element (s_i, d_i, w_i, t_i) to M, TCM adds the weight value w_i to the value of the bucket $M[h(s_i)][h(d_i)]$. With hash mapping, TCM can support various queries. However, hash collisions in the compressive matrix can result in unacceptable query accuracy. To alleviate this issue, TCM proposes to construct multiple matrices using different hash functions while returning the minimal result from one matrix. However, such a strategy is memory inefficient.

To reduce hash collisions, Gou *et al.* propose GSS [9] which utilizes a pair of fingerprints to represent an edge. In the $m \times m$ compressive matrix of GSS, each bucket maintains a fingerprint pair and a weight value for an edge. Suppose the hash range is [0, R) while R is power of two. For an element (s_i, d_i, w_i, t_i), GSS uses the F-bit ($F = \log_2 R/m$) suffixes of the hash values $h'(s_i)$ and $h'(d_i)$ to represent the fingerprint pair of the edge $s_i \rightarrow d_i$. The remaining $\log_2 m$-bit vectors of $h'(s_i)$ and $h'(d_i)$ are the row and column addresses to locate a bucket for storing the fingerprint pair and the weight. If the elements with different fingerprint pairs are mapped to the same bucket, GSS uses an extra buffer to store the later coming elements. The buffer length can seriously affect the query performance. To reduce the buffer size, GSS proposes a square hashing scheme to generate a fixed number of multiple alternative positions for the insertion of each edge.

Khan *et al.* [14] propose gMatrix, which contains multiple compressive matrices constructed by different reversible hash functions. The gMatrix can restore the node ID based on the address information but lead to more collision.

Existing graph stream summarization structures can support basic queries like edge/node weight aggregation query as well as more advance query like path reachability query. However, they suffer from poor query performance and accuracy due to the severe hash collisions raised by high-degree nodes. The further reason is that they use a uniform storage for both high-degree and low-degree nodes and map the edges with the same node to the same row or column of the compressive matrix. Scube also supports these queries with better performance.

Other efforts in graph streams include temporal range query processing [10], subgraph matching [3], [15]–[17], event detection [18], [19], concept drift detection [20], and regular path query [1]. Streaming graph processing also attracts recent interest [2], [21]–[23].

Traditional probability counting algorithms [24]–[27] address at estimating the cardinality of large data ensembles.

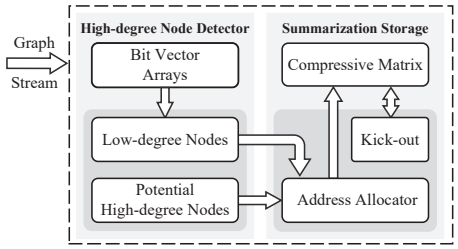

Fig. 4. Architecture of Scube

However, they are not applied to the problem of high-degree nodes identification in graph streams. How to efficiently identify high-degree nodes in a graph stream is a new problem.

III. The Design of Scube

The key to designing an efficient graph stream summarization structure is to identify the high-degree nodes in the graph stream and use a differentiated storage strategy for the associated edges. Accordingly, Scube proposes a novel low-probability events based probabilistic counting scheme and designs a dynamic address allocation scheme. Table I lists the notations used in the Scube design.

A. Overview

Figure 4 shows the architecture of Scube, which contains two components, a high-degree node detector and a summarization storage component. The high-degree node detector contains two arrays to estimate out-degree and in-degree. With low-probability events based probabilistic counting, each unit of the arrays consists of three parts: an L-bit vector initialized to zero for estimating the degree of a potential high-degree node, the number of times that the vector updates to $2^L - 1$ from zero, and the number of allocated addresses computed by the degree estimation. The summarization storage component is an $m \times m$ compressive matrix M. Each bucket in M maintains a pair of fingerprints of an edge and the associated weight value. Initially, Scube allocates two addresses for each node, *i.e.*, four positions of M for storing each edge. As high-degree nodes appear, Scube dynamically allocates more addresses and more positions of M to store the later coming edges with a high-degree node. When the optional buckets are all occupied, Scube uses a kick-out strategy to guarantee that all edges can be stored in M. Since the number of addresses calculated by the degree estimation may be less than that calculated by the exact degree, Scube explores a feedback mechanism to correct it. In the following, we present Scube's low-probability events based probabilistic counting and dynamic address allocation schemes, respectively.

B. Low-probability Events Based Probabilistic Counting

In this section, we first introduce the process of low-probability events based probabilistic counting. Then, we present the principle of our graph stream degree estimation scheme based on a proposed probability counting model. Finally, we analyze the influence of different parameters on the estimation accuracy.

Degree estimation in a graph stream is challenging in two aspects. First, maintaining the degree of all the nodes is costly in both memory and computation for a large-scale evolving graph stream. Second, it is difficult to eliminate the influence of duplicate edges which are common in a graph stream.

The probabilistic counting model. As shown in Fig. 5, for a node v in the graph stream, we perform the independent experiments on its neighbors $\{u_1, u_2, ..., u_k\}$ by computing the hash values of them. If the length of the binary code of the hash value is T, there are a total number of 2^T different binary codes. We use a limited set of patterns $P(T, i)$ to represent all non-zero T-bit hash codes. The expression of $P(T, i)$ is as follow,

$$P(T,i) = \{\underbrace{x \cdots x}_{T-i-1} 1 \underbrace{0 \cdots 0}_{i} | 0 \leq i \leq T-1, i \in N\} \quad (1)$$

where $\underbrace{0 \cdots 0}_{i}$ indicates that the consecutive i bits are all '0' and x represents '0' or '1'. For each hash value, the position of its first 1-bit determines which pattern it belongs to, and the probability of the pattern $P(T, i)$ is $\frac{1}{2^{i+1}}$. The probability of the pattern $P(T, i)$ with a larger i is smaller.

A basic idea to estimate the degree is using a T-bit vector initially set to zero to record the appearance of all patterns, *i.e.*, the i^{th} bit of the vector is set to '1' when the pattern $P(T, i)$ appears. Then we can deduce the degree expectation based on the probabilities of the patterns appeared. Intuitively, the degree expectation is larger when the vector records 1-bit in the higher digits. To filter most low-degree nodes, Scube only keeps the high digits of the vector to record the patterns with $i \geq R$ (low-probability events), where R is the position of the first recorded 1-bit of the hash value. Multiple low-probability events seldom appear simultaneously in a small number of neighbors of a low-degree node. Besides, such a method filters all the patterns with $i < R$ (frequent events), which significantly reduces the computation and memory costs. Parameter analysis experiment indicates that the average relative errors are almost the same when the length of the vector L exceeds one (we will show details about this with the experiment result shown in Fig. 6). To further reduce the

TABLE I
Notations in Scube

Notations	Descriptions
m	the width/depth of matrix M
$h(v)$	the hash value of node v
L	the length of bit vector
R	the position of the first recorded 1-bit of the hash value
c	the number of complete records
$\alpha[i]$	the i^{th} bit of a binary string α
$\varphi(\alpha)$	the first 1-bit of a binary string α
D_v	the exact degree of node v
$\tau(D_v)$	the degree estimation of node v
f_v	the fingerprint of node v
F	the length of fingerprint
N_v^0	the number of addresses allocated to source node v
N_v^1	the number of addresses allocated to destination node v
$A_v(n)$	the n^{th} address allocated to node v
K_v	the key of node v
B_1, B_2	the bit vector arrays in the detector
$B_1[i]$, $B_2[i]$	the i^{th} unit of B_1 and B_2
Z	the size of B_1/B_2
θ	the degree threshold of high-degree nodes

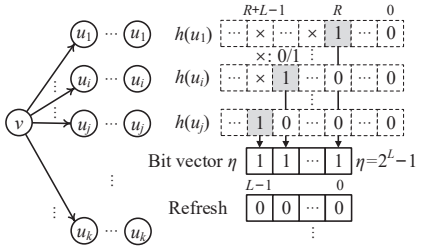

Fig. 5. Low-probability events based probabilistic counting

memory cost, Scube only maintains a bit vector with a small L finally. We call the process that the vector updates to 2^L-1 (all bits are '1') from zero *a complete record*. After each complete record, Scube resets the vector to zero, increases the number of complete records by one, and updates the number of addresses allocated to node v. The final degree estimation is obtained by multiplying the number of complete records by the degree expectation of a complete record and adding the estimation based on the final vector.

During one complete record, the duplicate edges only affect the degree estimation once because the corresponding bit is set to '1' no matter how many times the duplicate edges appear. If the duplicate edges appear in different complete records, it may affect the final degree estimation multiple times. An observation is that the weight values of the same edges are merged in the same bucket of the matrix M. Scube can determine whether the current edge is a duplicate edge by checking the matrix M. If the edge has existed in the matrix, it is a duplicate edge.

Figure 5 shows the degree estimation process. We elaborate on the out-degree estimation of a node v, while the in-degree estimation is similar. For each outgoing neighbor u_i of node v, Scube computes the T-bit hash value $h(u)$. Here, the hash value is just reused because inserting the edge into M needs to calculate the hash values. All the hash values except zero can be represented as the patterns $P(T,i)$. For convenience, we use $\alpha[i]$ to denote the i^{th} ($i \geq 0$) bit in the binary code of α with length T and introduce a function $\varphi(\alpha)$ to represent the first 1-bit of α. Formally,

$$\varphi(\alpha) = \begin{cases} \min\limits_{i\geq 0}\{\alpha[i]=1\} & \text{if } \alpha > 0 \\ T & \text{if } \alpha = 0 \end{cases} \tag{2}$$

For example, if $h(u)$ is 2 (*i.e.*, 10 in binary), $\varphi(h(u)) = 1$.

We use the notation η to denote the L-bit vector that is used to record the pattern of each $h(u)$ and use $\eta[i]$ to represent the i^{th} ($0 \leq i \leq L-1$) bit of η. The recording process is as follows. For each $h(u)$, Scube first checks whether the lowest R bits contain 1-bit by a shift operation, *i.e.*, $h(u)\&((1 << R)-1)$. If $(h(u)\&((1 << R)-1)) \neq 0$, Scube does nothing. Otherwise, Scube computes the first 1-bit of $h(u)$, *i.e.*, $\varphi(h(u))$. Only when $R \leq \varphi(h(u)) \leq R+L-1$ does Scube set the corresponding bit of η (*i.e.*, $\eta[\varphi(h(u))-R]$) to '1'. When all bits of η are '1' (*i.e.*, $\eta = 2^L-1$), Scube increases the number of complete records by one and updates the number of addresses allocated to v. Then, Scube refreshes the bit vector, *i.e.*, setting η to zero.

Under the assumption that the hash values of nodes are uniformly distributed, the probability of the pattern $P(L,i)$ is $(\frac{1}{2})^{i+1}$. For convenience, we use $D\triangle$ to denote the exact incremental degree of node v when η updates to 2^L-1 from zero. Formally, the estimation of $D\triangle$ is,

$$\tau(D\triangle) = 2^{R+L}\sum_{i=1}^{2^L-1}[(-1)^{v(i)+1} \times \frac{1}{i}] \tag{3}$$

where $v(i)$ denotes the number of 1-bit digits in the binary code of i. For example, if $i=3$ (i.e., 11 in binary), $v(i)=2$.

In Scube, when the degree of node v (denoted as D_v) exceeds a threshold θ (*i.e.*, the final degree estimation of node v exceeds θ), we regard node v as a high-degree node. The setting of θ is explained in Section III-C. For convenience, we use c to denote the number of complete records and use $\tau(D_v)$ to denote final degree estimation. Formally, $\tau(D_v) = c \times \tau(D\triangle) + \triangle'$, where $\triangle'$ represents the estimation based on final vector. Later we will explain that $\triangle'$ can be ignored under special parameters setting.

Probabilistic model analysis. Here, we give the derivation process of Eq. (3). For node v, we assume that the set of its distinct outgoing neighbors is $\{u_1, u_2, ..., u_n, ...\}$. The hash calculation process of each neighbor is independent. Initially, $\eta = 0$. Just after the record process of u_n, $\eta = 2^L-1$, while the exact degree increment is n. The occurrence conditions of $\eta = 2^L-1$ are: (I) $\varphi(h(u_n)) = k$ ($R \leq k \leq R+L-1$); (II) for each j which satisfies $R \leq j \leq R+L-1$ and $j \neq k$, there is at least one i ($1 \leq i \leq n-1$) such that $\varphi(h(u_i)) = j$; (III) for all i ($1 \leq i \leq n-1$), $\varphi(h(u_i)) \neq k$. The conditions contain a number of L cases because k has L different values.

We use the notation O to denote the set of all binary codes with T bits. Among all the elements of O, we define the following sets for each $i \geq 0$,

$$A_i = \{x|\varphi(x) = i\},\ B_i = \{x|\varphi(x) < i\},\ C_i = \{x|\varphi(x) > i\}$$

For an element x, the probability of $x \in A_i$ is,

$$Pr(A_i) = a_i = \frac{1}{2^{i+1}} \tag{4}$$

For each k ($R \leq k \leq R+L-1$), B_R, A_R, A_{R+1}, ..., A_k, C_k are disjoint, and their union is O. We arbitrarily extract $n-1$ elements from the above sets, and all possible situations form a set group termed as P_k^{n-1}. Formally,

$$P_k^{(n-1)} = (B_R \cup A_R \cup A_{R+1} \cup \cdots \cup A_k \cup C_k)^{(n-1)} \tag{5}$$

Among $P_k^{(n-1)}$, we use $Q_{k,L}^{(n-1)}$ to denote the set group containing all sets which satisfy conditions (II) and (III). Then,

$$\begin{aligned} Pr(\eta = 2^L-1) &= Pr(II\&\&III) \times Pr(I) \\ &= \sum_{k=R}^{R+L-1} Pr(Q_{k,L}^{(n-1)})a_k \end{aligned} \tag{6}$$

THEOREM 1. The probability that the first $n-1$ neighbors of v satisfy conditions (II) and (III), *i.e.*, $Pr(Q_{k,L}^{(n-1)})$ is,

$$\begin{aligned} &Pr(Q_{k,L}^{(n-1)}) = (1-a_k)^{n-1} + \\ &\sum_{t=1}^{L-1}(-1)^t \sum_{i\neq k, i_1<i_2<\cdots<i_t}^{R\leq i\leq R+L-1} (1-a_{i_1}-a_{i_2}-\cdots-a_{i_t}-a_k)^{n-1} \end{aligned} \tag{7}$$

Proof. We first discuss the case of $k = R$. When $L = 1$ we have

$$Q_{R,1}^{(n-1)} = (B_R \cup C_R)^{(n-1)} = (A_R)^{(0)} \tag{8}$$

where $(A_R)^{(0)}$ represents that none of the $n-1$ elements belong to set A_R. When $L = 2$ we have

$$\begin{aligned} Q_{R,2}^{(n-1)} &= (B_R \cup A_{R+1} \cup C_{R+1})^{(n-1)} - (B_R \cup C_{R+1})^{(n-1)} \\ &= (A_R)^{(0)} - (A_R \cup A_{R+1})^{(0)} \end{aligned} \tag{9}$$

When $L = 3$ we have

$$\begin{aligned} Q_{R,3}^{(n-1)} &= (B_R \cup A_{R+1} \cup A_{R+2} \cup C_{R+2})^{(n-1)} \\ &- (B_R \cup A_{R+1} \cup C_{R+2})^{(n-1)} \\ &- (B_R \cup A_{R+2} \cup C_{R+2})^{(n-1)} + (B_R \cup C_{R+2})^{(n-1)} \\ &= (A_R)^{(0)} - (A_R \cup A_{R+2})^{(0)} \\ &- (A_R \cup A_{R+1})^{(0)} + (A_R \cup A_{R+1} \cup A_{R+2})^{(0)} \end{aligned} \tag{10}$$

In general, by the principle of inclusion and exclusion,

$$\begin{aligned} Q_{R,L}^{(n-1)} &= (A_R)^{(0)} - \sum_{i=R+1}^{R+L} (A_i \cup A_R)^{(0)} \\ &+ \sum_{i<j}^{R<i,j\leq R+L} (A_i \cup A_j \cup A_R)^{(0)} \\ &- \sum_{i<j<l}^{R<i,j,l\leq R+L} (A_i \cup A_j \cup A_l \cup A_R)^{(0)} + \cdots \end{aligned} \tag{11}$$

Based on Eq. (4), we have

$$\begin{aligned} &Pr(Q_{R,L}^{(n-1)}) = (1-a_R)^{n-1} - \sum_{i=R+1}^{R+L-1} (1-a_i-a_R)^{n-1} \\ &+ \sum_{i<j}^{R<i,j\leq R+L-1} (1-a_i-a_j-a_R)^{n-1} - \cdots \\ &= (1-a_R)^{n-1} + \\ &\sum_{t=1}^{L-1} (-1)^t \sum_{i_1<i_2<\cdots<i_t}^{R<i\leq R+L-1} (1-a_{i_1}-a_{i_2}-\cdots-a_{i_t}-a_R)^{n-1} \end{aligned} \tag{12}$$

Generally, when $R \leq k \leq R+L-1$, replacing R with k into Eq. (12), and the theorem is proved. □

THEOREM 2. The expectation of the degree increment n satisfies,

$$E(n) = 2^{R+L} \sum_{i=1}^{2^L-1} (-1)^{v(i)+1} \frac{1}{i} \tag{13}$$

Proof. Based on Eq. (7), we have

$$\begin{aligned} &Pr(\eta = 2^L - 1) = \sum_{k=R}^{R+L-1} Pr(Q_{k,L}^{(n-1)}) a_k \\ &= \sum_i a_i (1-a_i)^{n-1} - \sum_{i<j} (a_i+a_j)(1-a_i-a_j)^{n-1} \\ &+ \sum_{i<j<l} (a_i+a_j+a_l)(1-a_i-a_j-a_l)^{n-1} - \cdots \end{aligned} \tag{14}$$

where all the indexes (i, j, l, ...) belong to $[R,\ R+L-1]$. Here, we introduce a function $\Upsilon(x)$, where $x \ll 1$,

$$\Upsilon(x) = x \sum_{i=1}^{\infty} i(1-x)^{i-1} \tag{15}$$

For $\Upsilon(x)$, we have

$$\begin{aligned} \Upsilon(x) &= x \sum_{i=1}^{\infty} i(1-x)^{i-1} = -x(\sum_{i=1}^{\infty} (1-x)^i)' \\ &\approx -x(\frac{1-x}{x})' = \frac{1}{x} \end{aligned} \tag{16}$$

where $(\frac{1-x}{x})'$ is the first derivative of $\frac{1-x}{x}$. The expectation of n can be expressed as

$$\begin{aligned} E(n) &= \sum_{i=1}^{\infty} i \times \sum_{k=R}^{R+L-1} Pr(Q_{k,L}^{(i-1)}) a_k \\ &= \sum_i \Upsilon(a_i) - \sum_{i<j} \Upsilon(a_i+a_j) - \sum_{i<j<l} \Upsilon(a_i+a_j+a_l) + \cdots \\ &= \sum_i \frac{1}{a_i} - \sum_{i<j} \frac{1}{a_i+a_j} + \sum_{i<j<l} \frac{1}{a_i+a_j+a_l} - \cdots \\ &= \sum_{t=1}^{L-1} (-1)^{(t+1)} \sum_{i_1<i_2<\cdots<i_t} \frac{1}{a_{i_1}+a_{i_2}+\cdots+a_{i_t}} \end{aligned} \tag{17}$$

where the indexes (i_1, i_2, ...) all belong to $[R,\ R+L-1]$. By changing the indexes to $l_j = R+L-1-i_j$, then $0 \leq l_j \leq L-1$, and $E(n)$ can be written as

$$\begin{aligned} &\sum_{t=1}^{L-1} (-1)^{t+1} \sum \frac{1}{a_{R+L-1-l_1}+a_{R+L-1-l_2}+\cdots+a_{R+L-1-l_t}} \\ &= \sum_{t=1}^{L-1} (-1)^{t+1} 2^{R+L} \sum \frac{1}{2^{l_1}+2^{l_2}+\cdots+2^{l_t}} \\ &= 2^{R+L} \sum_{i=1}^{2^L-1} [(-1)^{v(i)+1} \times \frac{1}{i}] \end{aligned} \tag{18}$$

where $v(i)$ denotes the number of 1-bit digits in the binary presentation of i. The theorem is thus proved. □

Parameters settings. Here, we discuss the impact of the parameters R, L, and c (the number of complete records) on the estimation accuracy. To this end, we conduct experiments to evaluate the *average relative error* (ARE) of low-probability events based probabilistic counting with different parameters settings. For convenience, we use $E_{R,L}$ to denote the increment expectation (Eq. (13)) with parameters R and L. We adjust three parameters (L, R, and c) to see how they respectively affect the ARE between the exact degree and the degree estimation $c \times E_{R,L}$. In the experiments, the number of complete records is updated to c just after the record of the last edge. In each setting, we test 10,000 times and plot the ARE distribution. Figure 6 shows the ARE is large when $L = 1$ and the AREs with different values of L ($L \geq 2$) are close. Figure 7 indicates that R has little effect on ARE. Figure 8 shows that the ARE decreases as c increases. Besides, when $c \geq 5$, the values of ARE are close.

Based on the above analysis, we set L to two in practice. On the one hand, when $L = 2$, the final estimation is equal to $c \times E_{R,2}$ regardless of whether the final vector is zero. As we mentioned before, we can ignore the estimation based on the final vector (*i.e.*, $\triangle'$) even if η contains one 1-bit when $L = 2$ because the occurrence of a low-probability event can be considered as an accident. On the other hand, when $c \times$

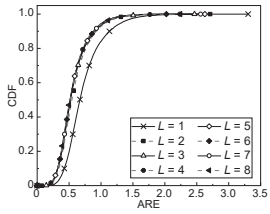

Fig. 6. ARE distribution with different values of L ($R = 10$ and $c = 1$)

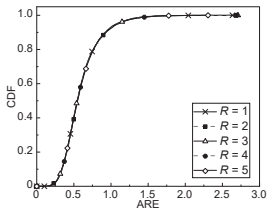

Fig. 7. ARE distribution with different values of R ($L = 2$ and $c = 1$)

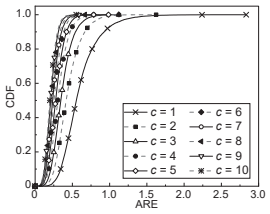

Fig. 8. ARE distribution with different values of c (R = 10 and L = 2)

$E_{R,L} = \theta$, a smaller L (*i.e.*, a smaller $E_{R,L}$) leads to a larger c, and thus ARE is smaller (Fig. 8). In this way, Scube updates the number of addresses only when c increases. Besides, we set R to satisfy that c is about five when $c \times E_{R,2} \approx \theta$.

C. Dynamic Address Allocation

As high-degree nodes appear, how to differentially store the high-degree and low-degree edges is not trivial. In existing graph stream summarization structures, *i.e.*, hash-based matrix, the edges with the same node are mapped to the same row or column. However, high-degree nodes have lots of neighbors, while low-degree nodes connect a few edges. A unified storage strategy cannot provide optimal performance for queries relevant to both high-degree and low-degree nodes. Scube designs a dynamic address allocation scheme on demand, which allocates different numbers of rows or columns to store low-degree edges and high-degree edges. In Scube, all the edges are stored in the $m \times m$ compressive matrix M.

Figure 9 shows the process of dynamic address allocation. Initially, all the nodes are regarded as low-degree nodes. Scube uses the linear congruence method [9], [28] to generate two row (resp. column) addresses for each source (resp. destination) node. A row address and a column address determine a position of M. In other words, Scube allocates four optional positions for inserting all the low-degree edges. At the same time, Scube uses an F-bit fingerprint to represent each node and marks each edge with a pair of fingerprints. The edges with the same low-degree source (resp. destination) node are all stored in two rows (resp. columns) of M. As the graph stream evolves, high-degree nodes appear, $i.e.$, their degrees exceed the threshold θ. We set the threshold to a certain proportion of the number of buckets in two rows/columns, $i.e.$, $\theta = 2\delta \times m$, where δ represents the proportion and $0 < \delta < 1$. We set δ to 0.8, which indicates that we extend the number of addresses when the previously allocated addresses are almost occupied. After the detector recognizes a high-degree source (resp. destination) node, Scube generates one more row (resp. column) address and allocates more positions to insert the

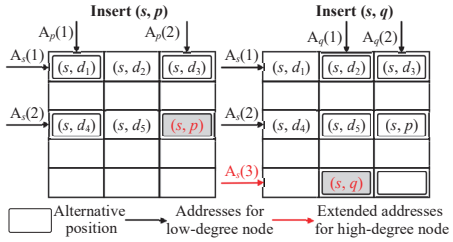

Fig. 9. Dynamic address allocation on demand

later edges with this node. Scube executes the same operations when the degree of a high-degree node increases by a certain percentage (δm). If all the corresponding buckets are occupied, Scube leverages a kick-out strategy to guarantee that all the edges can be hosted in M.

The specific address allocation is processed as follow. Scube first generates a series of seeds by the recursive formula,

$$S_v(n) = \begin{cases} f_v; & n = 1 \\ (a \times S_v(n-1) + i) \,\%\, c; & n \geq 2 \end{cases} \tag{19}$$

where a is a multiplier; i represents an increment constant; c is a module; f_v is the fingerprint of node v; $S(1)$ is the initial seed; and $\{S(n) \mid n \geq 1\}$ is the set of seeds. The address set of node v $\{A_v(n) \mid n \geq 1\}$ can be obtained by,

$$A_v(n) = \begin{cases} (h(v) >> F) \,\%\, m; & n = 0 \\ (A_v(0) + S_v(n)) \,\%\, m; & n \geq 1 \end{cases} \tag{20}$$

The address set is determined by $A_v(0)$ and f_v. In other words, different nodes generate different address sets. For convenience, we use N_v^0 and N_v^1 to denote the numbers of the addresses assigned to the source node v and the destination node v. For each element (s, d, w, t), if s and d are both low-degree nodes, Scube only performs two iterations for nodes s and d to obtain two row addresses $\{A_s(1), A_s(2)\}$ and two column addresses $\{A_d(1), A_d(2)\}$, resulting in four positions $\{(A_s(i),\ A_d(j)) \mid 1 \leq i \leq N_s^0, 1 \leq j \leq N_d^1\}$ ($N_s^0 = N_d^1 = 2$). Each position determines a bucket in M, *i.e.*, $M[A_s(i)][A_d(j)]$. If s (resp. d) becomes a high-degree node, *i.e.*, $c \times E_{R,L} > 2\delta \times m$, then $N_s^0 = 2+1$ (resp. $N_d^1 = 2+1$). Scube performs one more iteration to obtain one more address $A_s(3)$ (resp. $A_d(3)$) and inserts the edge to one empty bucket among the $N_s^0 \times N_d^1$ positions. In general, for each node v, Scube calculates the number of addresses allocated to v based on the degree estimation, *i.e.*, $N_v^0 = \lceil c \times E_{R,L}/(m\delta) \rceil$ or $N_v^1 = \lceil c \times E_{R,L}/(m\delta) \rceil$, then performs the same number of iterations to allocate addresses.

According to Eq. (19), we can obtain all the seeds $\{S_v(n) \,|\, n \geq 1\}$ based on the fingerprint f_v. Based on Eq. (20), Scube can restore the value of $A_v(0)$ by any address $A_v(n)$ and the corresponding iteration round n ($n \geq 1$). In other words, Scube can obtain all the addresses based on any address, the fingerprint, and the corresponding iteration round. This property is essential for the execution of the kick-out strategy. Therefore, Scube needs to store the two corresponding iteration rounds when inserting each edge to M. That is, if the edge $s \rightarrow d$ is stored in the bucket $M[A_s(i)][A_d(j)]$, Scube stores not only f_s and f_d but also i and j in this bucket.

During the insertion process of an element (s, d, w, t), all the corresponding buckets may be full. Inspired by cuckoo

hash [29], Scube leverages a kick-out strategy to guarantee that each edge can occupy one bucket. Scube can directly obtain all the addresses allocated to each edge stored in the matrix. However, how to choose an edge to kick requires careful consideration, which significantly affects the completion time of the subsequent insertion. The specific kick-out strategy is based on an observation: the newest two addresses assigned to both low-degree and high-degree nodes may be used in a low proportion. Therefore, the kicked edge is in one bucket of the four positions $\{(A_s(i),\, A_d(j)) \mid N_s^0 - 1 \le i \le N_s^0, N_d^1 - 1 \le j \le N_d^1\}$. Among these positions, Scube calculates the sum of the two iteration rounds stored in each position and chooses the edge with the smallest sum. A smaller sum represents a greater possibility of that the edge is inserted into the subsequent positions. If the subsequent buckets of the kicked edge are still full, Scube performs the similar process but excludes the last inserted edge to avoid loop kicks.

The number of allocated addresses computed for node v may be insufficient due to error. This situation leads to loop kicks because the number of v's neighbors is larger than the number of addresses multiplied by the number of buckets of one row/column. More importantly, most of the evicted edges contain node v. Based on the insight, Scube explores a feedback mechanism to correct the number of allocated addresses. Specifically, when the number of kicks exceeds a threshold, we observe the subsequent edges that kicked out and find the most frequent node. Then, we add its number of addresses by one in the detector and allocate one more address to it for subsequent insertion.

IV. Operations of Scube

In the construction of Scube, we integrate the above two methods into two core components including a high-degree node detector and a summarization storage structure which is a matrix M. Figure 10 shows the structure of the high-degree node detector. The detector contains two arrays B_1 and B_2 which are used to estimate the out-degree and in-degree of potential high-degree nodes, respectively. The sizes of B_1 and B_2 are both Z. Each unit in the arrays is composed of eight slots, which can be put in one cacheline. Based on mapping scheme, Scube maintains a key of a node, a two-bit vector, the number of complete records, and the number of addresses in each slot. We use K_v to represent the key of node v, and

$$K_v = (A_v(0) << F) + f_v \tag{21}$$

where $A_v(0) = (h(v) >> F)\,\%\, m$. Note that K_v determines the node collisions. In the following, we present the operations.

Insert. For each element (s, d, w, t) of a graph stream, Scube first obtains the number of addresses of nodes s and d (*i.e.*, N_s^0 and N_d^1) in the detector and computes the addresses based on Eqs. (19) and (20). Then Scube cooperates with the kick-out strategy to insert the edge into one of the corresponding buckets of M. If the edge is not a duplicate edge, Scube updates the out-degree of s and in-degree of d in the detector.

Specifically, for an element (s, d, w, t), Scube uses the F-bit suffixes of the hash values $h(s)$ and $h(d)$ to define the

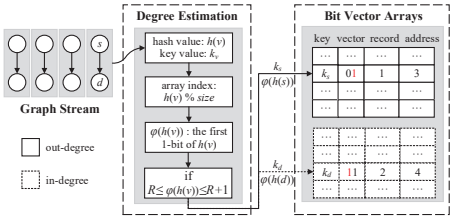

Fig. 10. High-degree node detector

fingerprints of nodes s and d, *i.e.*, f_s and f_d. Then, Scube generates N_s^0 row addresses, N_d^1 column addresses, and $N_s^0 \times N_d^1$ positions $\{(A_s(i),\, A_d(j)) \mid 1 \le i \le N_s^0, 1 \le j \le N_d^1\}$. Then, Scube checks the bucket in each position in order. If the bucket is empty, Scube stores the fingerprint pair (f_s, f_d), the weight w, and the two corresponding iteration rounds i and j in this bucket. Otherwise, if the fingerprint pair stored in the bucket is (f_s, f_d) and the iteration rounds are the same, Scube only adds w to the weight stored in this bucket. In this situation, the edge $s_i \to d_i$ is a duplicate edge.

If all the buckets are occupied, Scube performs the kick-out strategy to complete the insertion process. First, Scube checks the four positions $\{(A_s(i),\, A_d(j)) \mid N_s^0 - 1 \le i \le N_s^0, N_d^1 - 1 \le j \le N_d^1\}$ and evicts the edge from the bucket with the smallest sum of the two iteration rounds. Then, Scube stores the edge $s \to d$ in this bucket and inserts the evicted edge to an empty bucket of its subsequent positions. If these positions are still occupied, Scube continues to execute the kick-out operation. When performing the next kick operation, Scube excludes the edge inserted last time to avoid loop kicks.

If $s \to d$ is not a duplicate edge, Scube only updates the out-degree of s and the in-degree of d in the detector when $R \le \varphi(h(s)) \le R{+}1$ and $R \le \varphi(h(d)) \le R{+}1$, respectively. The update of out-degree is as follow, while that of in-degree is similarly. Scube maps s to the unit $B_1[K_s\%Z]$ and checks all the slots in it. If there is one slot that stores K_s, Scube performs the update operation, *i.e.*, Scube sets the $(\varphi(h(d)) - R)^{th}$ bit of the vector in this slot to '1' . When all the bits in the vector are '1', Scube refreshes the vector to zero, increases the number of complete records by one, and updates the number of addresses. Otherwise, Scube finds an empty slot to store K_s and updates the bit vector. If all the slots are occupied, Scube generates a new unit, links the new unit behind the unit, and stores K_s to one slot of the new unit.

Query. For convenience, we first introduce the process of obtaining the number of addresses in the detector. Given a key of a source (resp. destination) node v, *i.e.*, K_v, Scube locates to the unit $B_1[K_v\%Z]$ (resp. $B_2[K_v\%Z]$) and then checks all the slots in this unit. If this unit has linked units, Scube needs to check all the linked units. If any slot in these units has the same key K_v, Scube returns the number of addresses stored in this slot. Otherwise, node v is a low-degree node, and the number of allocated addresses is two. For an edge $s \to d$ stored in the matrix, Scube can directly obtain the number of addresses allocated to s and d because Scube can restore $A_s(0)$ and $A_d(0)$ and then computers K_s and K_d.

Graph stream queries contain two basic queries, the edge

Algorithm 1: EdgeQuery ($s \to d$)

```
1 obtain (f_s, f_d), K_s, K_d;
2 obtain N_s^0, N_d^1 in B_1[K_s%Z] and B_2[K_d%Z];
3 for i = 1 to N_s^0 do
4     for j = 1 to N_d^1 do
5         if M[A_s(i)][A_d(j)].fingerprintPair == (f_s, f_d) &&
            M[A_s(i)][A_d(j)].roundPair == (i, j) then
6             return M[A_s(i)][A_d(j)].weight.
7 return 0.
```

aggregation weight query and the node aggregation weight query. More complex queries are composed of the basic queries, such as the path reachability query and the subgraph aggregation weight query. Here, we introduce three typical queries, including the edge aggregation weight query (edge query for short), the node aggregation weight query (node query for short), and the path reachability query.

Edge query. Given an edge $s \to d$, this query returns the aggregated weight of the edge. Algorithm 1 shows the process of edge query. Scube first calculates the fingerprint pair (f_s, f_d) and the keys K_s and K_d based on the hash values $h(s)$ and $h(d)$ (line 1). Then, Scube obtains the number of allocated addresses of nodes s and d in the units $B_1[K_s\%Z]$ and $B_2[K_d\%Z]$, respectively (line 2). Next, Scube calculates the corresponding row addresses and column addresses based on Eqs. (19) and (20). In all buckets determined by the addresses, Scube checks the fingerprint pair, the two corresponding iteration rounds (lines 3-5). If any bucket matches, Scube returns the weight stored in the bucket (line 6). Otherwise, Scube returns zero, indicating the edge never appeared.

Node query. Given a source (resp. destination) node v, this query returns the aggregated weight of all outgoing (resp. incoming) edges of v. Similarly, Scube first computes the fingerprint f_v and the key K_v of node v based on $h(v)$. Then Scube seeks $B_1[K_v\%Z]$ (resp. $B_2[K_v\%Z]$) to obtain the number of allocated addresses of node v. Next, Scube calculates the corresponding row (resp. column) addresses and checks all buckets in these rows (resp. columns). For all buckets that store the same fingerprint of source (resp. destination) node and the corresponding iteration round, Scube aggregates the weight values of these buckets and returns the aggregated weight as the result. Algorithm 2 shows the process of source node query. Scube first obtains the fingerprint of node v and its number of addresses N_v^0 (line 2). Next, Sube checks all the buckets in these rows (lines 3-4). If the fingerprint of the source node is f_v and the iteration round matches, Scube adds the weight of the bucket to the result (lines 5-6). Finally, Scube returns the result.

Algorithm 2: SourceNodeQuery (v)

```
1 res = 0;
2 obtain f_s and N_v^0;
3 for i = 1 to N_v^0 do
4     for j = 1 to m do
5         if M[A_v(i)][j].sourceFingerprint == f_v &&
            M[A_v(i)][j].sourceRound == i then
6             res += M[A_s(i)][j].weight.
7 return res.
```

Path reachability query. Given a source node s and a destination node d, this query answers whether there is a path from s to d. The query process leverages the BFS algorithm. Specifically, Scube calculates the fingerprints f_s and f_d as well as the keys K_s and K_d. Then, Scube puts K_s into a queue. According to the *first-in-first-out* (FIFO) order, Scube removes the key at the head of the queue, computes the corresponding row addresses of the key, and marks it as visited. Then, Scube searches all the buckets in the corresponding rows. For any bucket that stores the same fingerprint calculated by the key and the corresponding iteration round, Scube restores the key of the destination node stored in this bucket. If the key is the same as K_d, Scube returns true as the result. When the key is not visited, Scube puts it into the queue. If the process is executed until the queue is empty, Scube returns false.

V. Analysis

A. Memory and Time Overhead Analysis

The memory overhead of Scube is $O(|M| + |B_1| + |B_2|)$. When constructing the matrix in Scube, we usually set $|M| \approx |E|$ to guarantee that the matrix can hold all the edges in a graph stream, where $|E|$ is the number of edges in the graph stream. In such a setting, the matrix still has many empty buckets to store more edges because the number of distinct edges is less than $|E|$. The memory of the high-degree node detector (*i.e.*, $|B_1| + |B_2|$) is tiny (about 1 MB in Fig. 28). Hence, the overall memory overhead is $O(|E|)$.

The insertion time overhead for each element (s, d, w, t) is $O(N_s^0 \times N_d^1 + 2 \times 1)$. During inserting, Scube needs to access one unit of the arrays B_1 and one unit of B_2 in the high-degree node detector and checks at most $N_s^0 \times N_d^1$ buckets in the matrix M. The overall insertion overhead is $O(1)$. Similarly, the computation overhead of the kick-out operation is $O(k)$, where k is the number of kicks.

The type of the query determines its computation overhead. Like the insertion overhead, the overhead of edge query is still $O(1)$. For a node query of source (resp. destination) node v, Scube needs to access a unit of B_1 (resp. B_2) and checks $N_v^0 \times m$ (resp. $N_v^1 \times m$) buckets in M. The time overhead of node query is $O(m)$. Path reachability query is equivalent to executing multiple node queries, and its time overhead is $O(m)$ in the best case and $O(m|V|)$ in the worst case.

B. Collision Analysis

Node collision. As explained in Section IV, the key's value determines the collision. Formally, for a source (resp. destination) node v, if there is at least another source (resp. destination) node u, which satisfies $K_v = K_u$, the source (resp. destination) node collision occurs. We use the notation S to denote the size of the key range, and $S = m \times 2^F$. The probability that the node v collides with other nodes is,

$$Pr(node) = 1 - (1 - \frac{1}{S})^{|V|} \approx 1 - e^{-\frac{|V|}{S}} \approx 1 - e^{-\frac{|V|}{m \times 2^F}} \quad (22)$$

where $|V|$ represents the number of nodes in the graph stream. When we set F to 16, the probability of node collision is small.

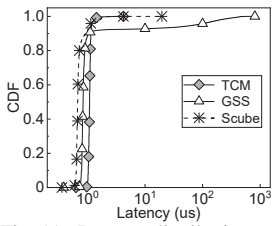

Fig. 11. Latency distribution of edge queries on wiki-talk

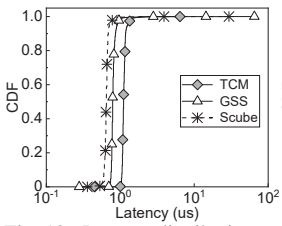

Fig. 12. Latency distribution of edge queries on stack

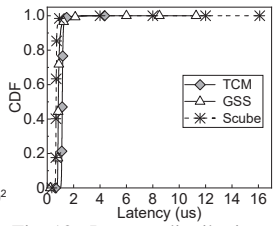

Fig. 13. Latency distribution of edge queries on dbpedia

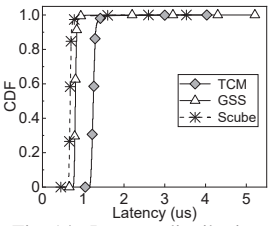

Fig. 14. Latency distribution of edge queries on net-flow

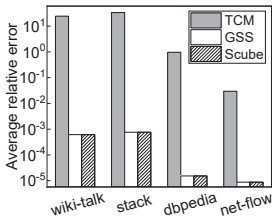

Fig. 15. ARE of edge queries on the four datasets

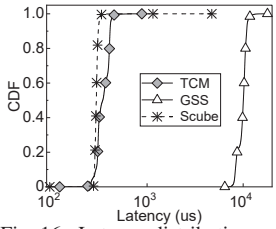

Fig. 16. Latency distribution of destination node queries on wiki-talk

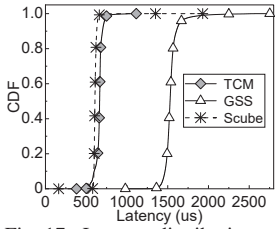

Fig. 17. Latency distribution of destination node queries on stack

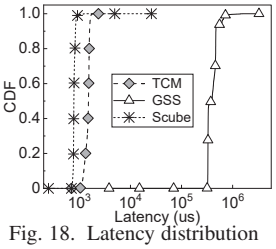

Fig. 18. Latency distribution of destination node queries on dbpedia

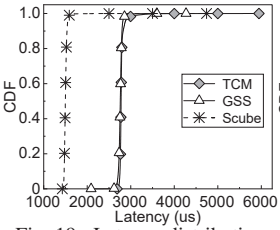

Fig. 19. Latency distribution of destination node queries on net-flow

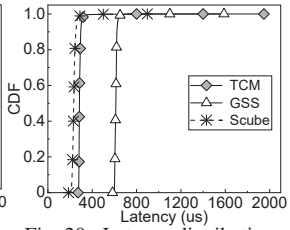

Fig. 20. Latency distribution of source node queries on net-flow

Edge collision. Formally, for an edge $e = s \to d$, if there is at least another edge $v \to u$ ($s \neq v$ or $d \neq u$), which satisfies $K_s = K_v$ and $K_d = K_u$, edge collision happens. We use X to represent the set of all the edges with the source node s or the destination node d. The probability that the edges of set X collide with e is $1/S$ while the probability that the other edges which are not in the set collide with e is $1/S^2$. The probability that e suffers from edge collision is,

$$\begin{aligned} Pr(edge) &= 1 - (1 - \frac{1}{S^2})^{|E|-|X|} \times (1 - \frac{1}{S})^{|X|} \\ &\approx 1 - e^{-\frac{|E|+(S-1)|X|}{S^2}} \approx 1 - e^{-\frac{|E|+(S-1)|X|}{m^2 \times 2^{2F}}} \end{aligned} \tag{23}$$

Since $|X|$ is small, we can see that the probability of edge collision is much smaller than that of node collision.

VI. Performance Evaluation

A. Methodology

We implement Scube and make the source code publicly available [1]. We compare Scube with the baselines GSS [9] and TCM [8].

Datasets. In the experiment, we use four large-scale real-world graph stream datasets.Table II summarizes the statistics of the datasets.

(1) Wiki-talk [30]. This dataset is the communication network of the English Wikipedia. Nodes are users, and edges represent user interactions. The dataset contains 2,987,535 nodes and 24,981,163 edges.

(2) Stackoverflow [31] (stack for short). It contains the interaction data from the StackExchange site "Stack Overflow". The nodes are users, and edges represent users' answers to others' questions. The dataset contains $2,601,977$ users and $63,497,050$ interactions.

(3) Dbpedia [12]. This is the hyperlink network of Wikipedia. Nodes represent pages in Wikipedia and edges correspond to hyperlinks. Dbpedia contains 18,268,991 nodes and 172,183,984 edges.

[1]https://github.com/CGCL-codes/Scube

(4) Network flow [32] (net-flow for short). It contains partial anonymous traffic traces from CAIDA's monitor. Each trace represents an IP address sending a packet to another while the packet size is the weight. The dataset contains $2,121,486$ nodes and $403,436,907$ edges.

In the experiment, we set the sizes of matrices of Scube, TCM, and GSS to the same. In TCM, we configure six matrices for it. In Scube and GSS, we set the fingerprint size F to 16 bits. Meanwhile, we set the width/depth of the matrix to be smaller and split each bucket of the matrix into two slots, which can alleviate mapping conflicts and improve insertion throughput. Each slot stores the information of one edge, and the matrix size satisfies $|M| \approx |E|$. In GSS, we set the address number of each node to four, *i.e.*, 16 alternative positions for each edge. In the high-degree node detector, we set the size of the two arrays (*i.e.*, Z) to a prime number. We use 32 bits to represent the key of each node. To put the vector, the number of complete records, and the number of addresses into one field, we use two bits, eight bits, and six bits, respectively. Moreover, we set the degree threshold θ to $3.2 \times m$.

Metrics. Average query latency measures the average latency of all the queries. Insertion throughput measures the number of edges inserted per second. Memory cost measures the memory consumption of the entire structure. Average relative error measures the accuracy of query results in edge and node queries. Given the query result $\widetilde{Q}$ and the the exact value Q, the *relative error* (RE) can be computed by $\frac{\widetilde{Q}-Q}{Q}$. The ARE is the average of all the RE values in a query set.

We also evaluate the *false positive rate* (FPR) of Scube. For

TABLE II
Datasets

Datasets	nodes	edges	size (GB)
Wiki-talk	2,987,535	24,981,163	0.57
Stack	2,601,977	63,497,050	1.66
Dbpedia	18,268,991	172,183,984	2.52
net-flow	2,121,486	403,436,907	13

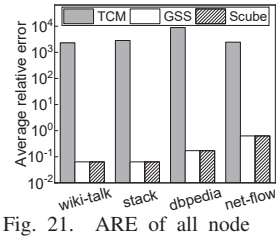

Fig. 21. ARE of all node queries on the four datasets

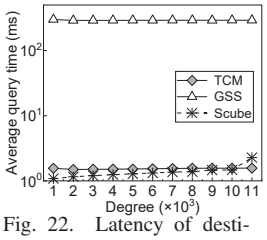

Fig. 22. Latency of destination node queries with degree division on dbpedia

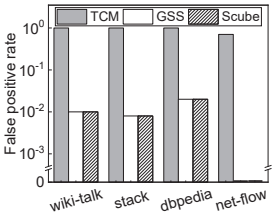

Fig. 23. FPR of path reachability queries on the four datasets

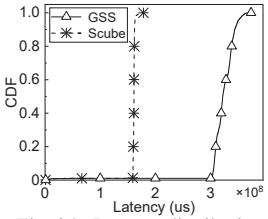

Fig. 24. Latency distribution of path reachability queries on wiki-talk

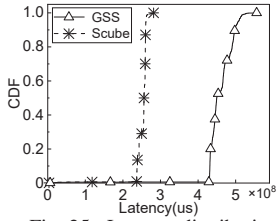

Fig. 25. Latency distribution of path reachability queries on stack

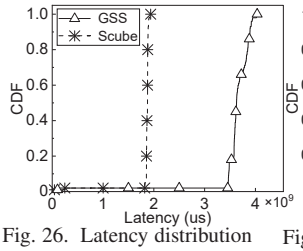

Fig. 26. Latency distribution of path reachability queries on dbpedia

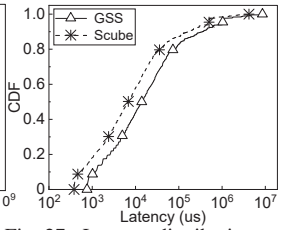

Fig. 27. Latency distribution of path reachability queries on net-flow

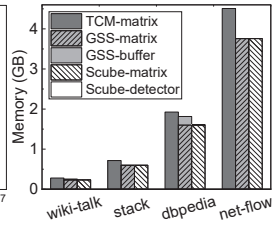

Fig. 28. Memory cost on the four datasets

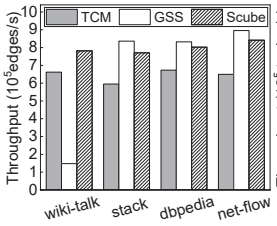

Fig. 29. Insertion throughput on the four datasets

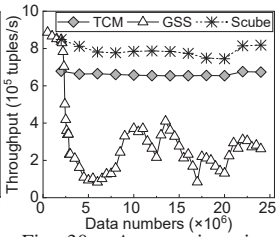

Fig. 30. Average insertion throught on wiki-talk

a path reachability query that the given destination node d is reachable from the given source node s, Scube must return true. For a path reachability query that there is no path from s to d, we call it a non-reachability query. The result of a non-reachability query may return true, and the FPR is $\frac{N}{|\vartheta|}$, where ϑ is a set of non-reachability queries and N is the number of queries that return true. We also evaluate FPR of the high-degree node detector in Scube. In the measurement, $|\vartheta|$ is the number of low-degree nodes in a dataset, and N is the number of low-degree nodes identified as high-degree nodes. *Hit rate* (HR) measures the accuracy of the high-degree node detector. It is computed by $\frac{N}{|\vartheta|}$, where $|\vartheta|$ is the number of high-degree nodes in a dataset and N is that of high-degree nodes correctly identified.

We conduct four groups of experiments. The first group evaluates the query performance of three typical queries. In edge queries and node queries, we test all the edges and nodes in the corresponding datasets and measure the latency and ARE. In the set of path reachability queries, all the queries are non-reachability queries, and the set cardinality is $10,000$. We measure the average latency and FPR. The second group evaluates the memory cost. The third group evaluates the insertion throughput. The fourth group evaluates the impact of the detector and the kick-out strategy on insertion time as well as the HR of the high-degree node recognition. We conduct the experiments on a machine with a 16-core 2.4GHz Xeon CPU, 64 GB RAM, and 1TB HDD.

B. Results

Figures 11-14 show the latency distributions of edge queries on the datasets wiki-talk, stack, dbpedia, and net-flow, respectively. The results show Scube performs best in edge queries. Figure 11 shows Scube reduces the tail latency of GSS by more than one order of magnitude.

Figure 15 plots ARE of edge queries on the four datasets. Scube significantly reduces ARE of edge queries by several orders of magnitude compared to TCM, even though we construct six matrices for TCM. AREs of Scube and GSS are the same because they use the same hash function and set the same fingerprint length.

Figures 16-19 illustrate the latency distributions of destination node queries on different datasets. Scube reduces the average query latency of GSS by 50%-99.8% and reduces the latency of TCM by 50%. Figure 20 plots the latency distribution of the source node queries on net-flow. Similarly, Scube reduces the latency of GSS by 64.8%. The latency of the source node query is much smaller than that of the destination node query because the locality of accessing the buckets in rows is better than that in columns. Figure 21 shows ARE of node queries (including all the source and destination node queries) in the four datasets. Scube and GSS have high accuracy for node queries, while the accuracy of node queries in TCM is unacceptable.

To examine how the node degree influences the performance of node queries, we divide all the queries into multiple collections according to the node degrees with a granularity of 1,000. Figure 22 depicts the latency of destination node queries with degree division on dbpedia. Scube reduces the average latency of GSS by more than two orders of magnitude regardless of the degree. Moreover, we can see that the latency of Scube increases very slowly because the number of allocated addresses increases as the degree increases.

Figure 23 shows FPRs of the non-reachability queries on different datasets. FPRs of Scube and GSS are very low while that of TCM is almost 100%. In other words, the accuracy of the path reachability queries is very high in Scube, while TCM cannot support path reachability queries.

Figures 24-27 plot the latency distribution of the non-reachability queries on the four datasets respectively. Here, we only present the results of Scube and GSS, because TCM is useless for path reachability queries. Scube reduces the average latency of GSS by 48.5%-53.3%.

Figure 28 shows the memory costs on the four datasets. The memory cost of Scube is close even lower to that of GSS

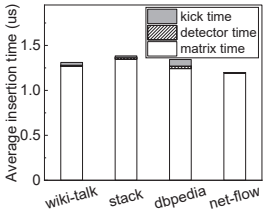

Fig. 31. Average insertion time breakdown in Scube

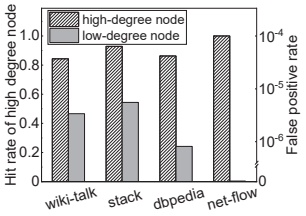

Fig. 32. The hit rate and the FPR of the high-degree node detector

because the buffer size of GSS is large in the sharply skewed dataset. In the figure, the histogram for scube-detector is barely visible, because it is very small (about 1MB).

Figure 29 presents the insertion throughput on different datasets. Figure 30 shows the average throughput during the insertion of the dataset wiki-talk. The throughputs of Scube, TCM, and GSS are close in the datasets stack, dbpedia, and net-flow. In the sharply skewed dataset wiki-talk, a large number of high-degree nodes lead to the increase in the buffer size of GSS. Therefore, the throughput of GSS drops sharply. Since Scube stores all the edges into the compressive matrix through the dynamic address allocation strategy, its throughput is less affected by high-degree nodes.

Figure 31 breakdowns the average insertion time of Scube. The insertion time contains three parts, the time to access the detector, the time of insertion in the matrix, and the time to perform the kick-out strategy. The result shows the detector time and the kick time are very short. Figure 32 shows HRs and FPRs in the high-degree node detector of Scube on the four datasets. The results indicate that the identification accuracy of high-degree nodes is high while FPR is very low. This demonstrates that our low-probability events based probabilistic counting is reliable.

VII. Conclusion

In this paper, we show that the key to summarizing skewed graph streams is to identify high-degree nodes and use differentiated storage strategies for the associated edges. We propose Scube, an efficient summarization structure for skewed graph streams. Scube proposes a novel probabilistic counting scheme to identify the high-degree nodes in a graph stream and dynamically allocates more addresses for storing the associated edges. Experiment results demonstrate the superiority of Scube compared to state-of-the-art designs.

VIII. Acknowledgements

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

References

[1] A. Pacaci, A. Bonifati, and M. T. Özsu, "Regular path query evaluation on streaming graphs," in *Proceedings of SIGMOD,* Portland, OR, USA, June 14-19, 2020.

[2] P. Vaziri and K. Vora, "Controlling memory footprint of stateful streaming graph processing," in *Proceedings of ATC,* Virtual Event, 2021.

[3] Y. Li, L. Zou, M. T. Özsu, and D. Zhao, "Time constrained continuous subgraph search over streaming graphs," in *Proceedings of ICDE,* Macao, China, April 8-11, 2019.

[4] A. C and K. Mahesh, "Graph analytics applied to COVID19 karnataka state dataset," in *Proceedings of ICISS,* Edinburgh, UK, March 17-19, 2021.

[5] H. Chen, H. Jin, and S. Wu, "Minimizing inter-server communications by exploiting self-similarity in online social networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 4, pp. 1116–1130, 2016.

[6] "WeChat statistics." https://expandedramblings.com/index.php/wechat-statistics, 2021.

[7] "Tencent Health Code." https://www.tencent.com/zh-cn/business/health-code, 2021.

[8] N. Tang, Q. Chen, and P. Mitra, "Graph stream summarization: From big bang to big crunch," in *Proceedings of SIGMOD,* San Francisco, CA, USA, June 26 - July 01, 2016.

[9] X. Gou, L. Zou, C. Zhao, and T. Yang, "Fast and accurate graph stream summarization," in *Proceedings of ICDE,* Macao, China, April 8-11, 2019.

[10] M. Chen, R. Zhou, H. Chen, J. Xiao, H. Jin, and B. Li, "Horae: A graph stream summarization structure for efficient temporal range query," in *Proceedings of ICDE,* Kuala Lumpur, Malaysia, May 9-12, 2022.

[11] T. Ying, H. Chen, and H. Jin, "Pensieve: Skewness-aware version switching for efficient graph processing," in *Proceedings of SIGMOD,* Portland, OR, USA, June 14-19, 2020.

[12] "DBpedia," http://konect.cc/networks/dbpedia-link, 2021.

[13] M. Mariappan, J. Che, and K. Vora, "Dzig: sparsity-aware incremental processing of streaming graphs," in *Proceedings of EuroSys,* UK, April 26-28, 2021.

[14] A. Khan and C. C. Aggarwal, "Query-friendly compression of graph streams," in *Proceedings of ASONAM,* CA, USA, August 18-21, 2016.

[15] K. Kim, I. Seo, W. Han, J. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong, "Turboflux: A fast continuous subgraph matching system for streaming graph data," in *Proceedings of SIGMOD,* Houston, TX, USA, June 10-15, 2018.

[16] C. Wang and L. Chen, "Continuous subgraph pattern search over graph streams," in *Proceedings of ICDE,* Shanghai, China, March 29 - April 2, 2009.

[17] X. Gou and L. Zou, "Sliding window-based approximate triangle counting over streaming graphs with duplicate edges," in *Proceedings of SIGMOD,* Xian, China, June 20-25, 2021.

[18] C. Song, T. Ge, C. X. Chen, and J. Wang, "Event pattern matching over graph streams," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 413–424, 2014.

[19] M. H. Namaki, K. Sasani, Y. Wu, and T. Ge, "BEAMS: bounded event detection in graph streams," in *Proceedings of ICDE,* San Diego, CA, USA, April 19-22, 2017.

[20] R. Paudel and W. Eberle, "An approach for concept drift detection in a graph stream using discriminative subgraphs," *ACM Transactions on Knowledge Discovery from Data*, vol. 14, no. 6, pp. 70:1–70:25, 2020.

[21] M. Mariappan, J. Che, and K. Vora, "Dzig: sparsity-aware incremental processing of streaming graphs," in *Proceedings of EuroSys,* UK, April 26-28, 2021.

[22] G. Feng, Z. Ma, D. Li, S. Chen, X. Zhu, W. Han, and W. Chen, "Risgraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s," in *Proceedings of SIGMOD,* Xi'An, China, June 20-25, 2021.

[23] A. P. Iyer, Q. Pu, K. Patel, J. E. Gonzalez, and I. Stoica, "TEGRA: efficient ad-hoc analytics on evolving graphs," in *Proceedings of NSDI,* April 12-14, 2021.

[24] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *Journal of Computer and System Sciences*, vol. 31, no. 2, pp. 182–209, 1985.

[25] K. Whang, B. T. V. Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Transactions on Database Systems*, vol. 15, no. 2, pp. 208–229, 1990.

[26] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *Proceedings of ESA,* Budapest, Hungary, September 16-19, 2003.

[27] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm," in *Proceedings of AOFA,* Juan-les-pins, France, June 17-22, 2007.

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

[29] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[30] "wiki-talk," http://konect.cc/networks/wiki_talk_en, 2021.

[31] "Sx-stackoverflow," http://konect.cc/networks/sx-stackoverflow, 2021.

[32] "Caida," https://www.caida.org/data/passive/passive_dataset.xml, 2021.