# Horae: A Graph Stream Summarization Structure for Efficient Temporal Range Query

Ming Chen†, Renxiang Zhou†, Hanhua Chen†, Jiang Xiao†, Hai Jin†, Bo Li‡
†*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, Wuhan, 430074, China*
‡*Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, China*
{mingc, mr_zhou, chen, jiangxiao, hjin}@hust.edu.cn, bli@cse.ust.hk

***Abstract*—Graph stream, referred to as an evolving graph with a timing sequence of updated edges through a continuous stream, is an emerging data format widely used in big data applications. Coping with a graph stream is challenging because: 1) fully storing the continuously produced and extremely large-scale datasets is difficult if not impossible; 2) supporting queries relevant to both graph topology and temporal information is nontrivial. Recently, graph stream summarization techniques have attracted much attention in providing approximate storage and query processing for a graph stream. Existing designs largely utilize hash functions to reduce the graph scale and leverage a compressive matrix to represent the graph stream. However, such designs are unable to store the time dimension information of graph streams, and thus fail to support temporal queries.**

**In this paper, we propose Horae, a novel graph stream summarization structure for efficient temporal range query, which presents a time prefix embedded multi-layer summarization structure. Our design is based on the insight that an arbitrary temporal range of length $L$ can be decomposed to at most $2 \log L$ sub-ranges, where all the time points in each sub-range have the same binary code prefix. We further design an efficient *Binary Range Decomposition* (BRD) algorithm, which achieves a logarithmic scale query processing time. Experimental results show that Horae significantly reduces the latency of various temporal range queries by two to three orders of magnitude compared to the state-of-the-art designs.**

***Index Terms*—Graph stream, temporal range query**

## I. Introduction

The emerging graph stream [1]–[4] represents an evolving graph formed as a timing sequence of elements (updated edges) through a continuous stream. Each element in a graph stream is formally denoted as $(s_i, d_i, w_i, t_i)$ $(i \geq 0)$, meaning the directed edge of a graph $G = (V, E)$, *i.e.*, $s_i \rightarrow d_i$ $(s_i \in V,\ d_i \in V,\ s_i \rightarrow d_i \in E)$, is produced at time $t_i$ with a weight value $w_i$. An edge can appear multiple times at different time instants with different weights. Such a general data form is widely used in big data applications [2], such as user behavior analysis in social networks [5], close contact tracking in epidemic prevention [6], and vehicle surveillance in smart cities [7].

Real-world big data applications can create tremendously large-scale graph stream data [8]. For example, 902 million WeChat daily active users generate 38 billion messages each day [9]. Tencent Health Code covers more than one billion people and has generated 24 billion health code scans (needed when an individual enters important places, *e.g.*, hospital or airport, during COVID-19) [10]. The enormous data scale makes the management of graph streams extremely challenging, especially in the aspects of (1) storing the continuously produced and large-scale datasets, and (2) supporting queries relevant to both graph topology and temporal information.

To address these issues, recent research has mainly focused on graph stream summarization techniques [1], [11] which aim at achieving practicable storage and supporting various queries relevant to graph topology at the cost of slight accuracy sacrifice. Tang *et al.* propose TCM [1], which uses an $m \times m$ $(m \ll |V|)$ compressive adjacency matrix $M$ to represent a graph stream. Each bucket in the matrix maintains a weight value initially set to zero. For an element $(s_i,\ d_i,\ w_i,\ t_i)$ in the graph stream, TCM adds $w_i$ to the weight value of the bucket at the $h(s_i)^{th}$ row and the $h(d_i)^{th}$ column, *i.e.*, $M[h(s_i)][h(d_i)]$, where $h(\cdot)$ is a hash function with the value range $[0, m)$. Accordingly, TCM can support Boolean and aggregation queries. For example, we can query whether an edge $x \rightarrow y$ exists by checking against the matrix. If $M[h(x)][h(y)] = 0$, $x \rightarrow y$ definitely never appeared in the graph stream; otherwise, it appeared with a high probability. The accumulated weight value of $x \rightarrow y$ can also be obtained as $M[h(x)][h(y)]$. We can further achieve the aggregated weight of all the outgoing edges of $x$ (node aggregation weight query) by computing $\sum_{i=0}^{m-1} M[h(x)][i]$. To improve the query precision based on a compressive matrix, Gou *et al.* propose GSS [11] to alleviate the impact of hash collisions by fingerprinting each edge. However, the storage of fingerprint requires extra memory cost.

Graph summarization structures are useful in different application scenarios. For example, in smart city vehicle surveillance, the volume of captured video is prohibitively huge for long-term storage. We can recognize vehicle license plates from the captured video and convert the video stream into a succinct graph stream with each element like ($ACX$-7581, $camera$ #27, 1, 07:01am 01.01.2020). With graph stream summarization, we can easily answer the query "was the car $ACX$-7581 ever captured by the $camera$ #27?" In a data center, communications between servers and clients can also

*The Corresponding Author is Hanhua Chen (chen@hust.edu.cn).

be represented by a graph stream summarization structure. We can obtain the upload traffic of a server with a node aggregation weight query.

However, existing summarization structures are unable to store the temporal information in a graph stream and thus fail to support temporal queries. It is difficult to answer such queries: "was the car $ACX-7581$ ever captured by the $camera$ #27 between 07:01am 01.01.2021 and 23:59pm 02.02.2021?" and "what is the total upload traffic volume of the server #2812 between 15:00pm and 16:00pm?"

It is nontrivial to design a scheme to support such temporal range queries over graph streams. On the one hand, storing the temporal information with an existing graph stream summarization structure in a straightforward way can raise high query latency for a temporal range query. Specifically, for an element ($s_i$, $d_i$, $w_i$, $t_i$), one can add $w_i$ to the value of the bucket $M[h(s_i \mid t_i)][h(d_i \mid t_i)]$, where '|' represents the concatenation operation of two strings. Accordingly, a simple temporal point query "did the edge $x \rightarrow y$ appear at time $t$?" can be answered by checking $M[h(x \mid t)][h(y \mid t)]$. However, a temporal range query cannot be efficiently evaluated. Formally, a pair of time points $t_b$ and $t_e$ ($t_b \leq t_e$) can specify a temporal range $[t_b, t_e]$ with the length $t_e - t_b + 1$. With the existing structures (*e.g.*, TCM), answering such a query "did the edge $x \rightarrow y$ appear within the time range $[t_b, t_e]$?" needs to discretely evaluate the temporal point query against the structure $t_e - t_b + 1$ times. The query processing latency increases linearly with the length of the range, which can be prohibitively costly in practice.

On the other hand, traditional data structures, *e.g.*, the interval tree [12], [13] and the segment tree [14], [15], cannot address the temporal range query over graph streams. Given a predefined set of intervals, the interval tree recursively builds a binary tree based on the median of the intervals' endpoints, where each tree node stores an interval of the set. The interval tree can only query the data of the given intervals rather than the data of an arbitrary interval. For a certain range boundary, the segment tree first builds a static balanced tree in a bottom-up manner, where the interval of a non-leaf node is the union of the intervals of its two children and the interval of the root is the range boundary. It then inserts the data of each point to the nodes where the intervals cover the point. The segment tree provides arbitrary range query within the boundary by aggregating the data of several intervals. However, the segment tree cannot be applied to the dynamic graph stream which is infinite in time dimension and cannot guarantee the data integrity of each level in a one-pass manner. This thus can result in unacceptable query latency.

In this work, we propose Horae, a novel graph stream summarization structure to efficiently support temporal range queries. By exploring a time prefix embedded multi-layer summarization structure, Horae can effectively handle a temporal range query of an arbitrary range length $L$ with a worst query processing time of $O(\log L)$. The basic idea of Horae's time prefix embedded multi-layer summarization structure is as follows. An arbitrary temporal range of length $L$ can be decomposed to at most $2 \log L$ sub-ranges, where all the time points in each sub-range have the same binary code prefix. For example, $[t_8, t_{13}] = [t_8, t_{11}] + [t_{12}, t_{13}]$, where all the time points between $t_8$ (*i.e.*, 1000) and $t_{11}$ (*i.e.*, 1011) have the same common prefix '10', while all the time points between $t_{12}$ (*i.e.*, 1100) and $t_{13}$ (*i.e.*, 1101) have the same prefix '110'. Here, we define the *prefix size* as the number of binary digits in the common prefix (*e.g.*, the prefix size of the common prefix '10' is two while that of '110' is three).

A Horae structure contains a number of $l = \lceil \log(t_u+1) \rceil + 1$ layers, where $t_u$ is the current time point of a graph stream. To cope with the infinity in the time dimension, the number of layers in Horae dynamically increases as $t_u$ grows. Horae arranges the layers according to different prefix sizes. Each layer leverages a matrix to store the complete graph stream data aggregated by the sub-ranges with the same prefix size. Consider the example with $t_u = t_7$, Horae has four layers. The first layer contains eight sub-ranges $\{[t_0]$ ([0000]), $[t_1]$ ([0001]), ..., $[t_7]$ ([0111])$\}$ with prefix size of four; the second layer contains four sub-ranges $\{[t_0, t_1]$ ([0000, 0001]), $[t_2, t_3]$ ([0010, 0011]), ..., $[t_6, t_7]$ ([0110, 0111])$\}$ with prefix size of three, and so on. Formally, the $p^{th}$ layer aggregates the data by the sub-ranges $\{[q \cdot 2^{p-1}, (q+1) \cdot 2^{p-1} - 1]$ $(q \geq 0)\}$ with prefix size $l - p + 1$. The sub-ranges of each layer have the same prefix size, *i.e.*, the same range length. In a nutshell, each layer of the structure represents a summarization of a graph stream with carefully selected granularity (corresponds to a prefix size). During the construction of each layer, Horae concatenates each edge and the time prefix of the corresponding size for inserting the edge information into the corresponding matrix. Similarly, Horae concatenates an edge/node and the sub-range prefix to perform a sub-range query.

To efficiently evaluate temporal range queries on top of the Horae structure, we further design a novel *Binary Range Decomposition* (BRD) algorithm. The BRD algorithm decomposes a temporal range query with an arbitrary length $L$ into at most $O(\log L)$ sub-range queries against different layers of the structure. Therefore, Horae reduces the query processing time to a logarithmic scale. Experimental results show that Horae reduces the latency of temporal range queries by two to three orders of magnitude compared to existing designs.

## II. Related Work

To manage the large-scale unbounded data [1], [2], [11], [16], [17], graph stream summarization techniques have recently been proposed [1], [11], [18]. Existing designs store the weight and topology information in the graph stream by leveraging space efficient hash-based matrices and they can support different kinds of queries, such as edge/node existence queries and edge/node aggregation queries. However, little is known about how to exploit the temporal information and support efficient temporal range queries.

Tang *et al.* propose TCM [1], which uses an $m \times m$ hash-based adjacency matrix to represent a graph stream. Each bucket in the matrix maintains a weight value, which is initially set to zero. When a new element ($s_i$, $d_i$, $w_i$, $t_i$) in the graph stream is produced, TCM directly adds $w_i$ to the value

of the bucket located at the $h(s_i)^{th}$ row and $h(d_i)^{th}$ column. If different edges are mapped to a same bucket, their weight values are merged in the bucket. TCM can support edge/node existence queries and edge/node aggregation queries. However, since $m \ll |V|$, TCM suffers from hash collisions and unsatisfied query precision. One can extend TCM to a multi-matrix structure using different hash functions and achieve more accurate results with additional memory.

To alleviate hash collisions, Gou *et al.* propose GSS [11] which identifies an edge with a pair of fingerprints. GSS essentially uses a hash function $h'(\cdot)$ with a larger range size $R$ ($R \gg m$). Assume $R$ and $m$ are powers of two. For the element ($s_i$, $d_i$, $w_i$, $t_i$), GSS uses the $\log_2 m$-bit prefixes of the hash values $h'(s_i)$ and $h'(d_i)$ as the row address and the column address to locate a bucket for inserting the edge information. The remaining $F$-bit vectors ($F = \log_2(R/m)$) of $h'(s_i)$ and $h'(d_i)$ form a pair of fingerprints of $s_i \rightarrow d_i$. Each bucket of the matrix in GSS stores a pair of fingerprints of an edge and the associated weight. During edge inserting, if the corresponding bucket is empty, GSS inserts both $w_i$ and the fingerprint pair into the bucket. Otherwise, GSS checks the fingerprint pair stored in the bucket. If matched, GSS adds $w_i$ to the weight stored in the bucket. Otherwise, GSS inserts the edge into an extra buffer, *i.e.*, an adjacency list. In addition, GSS uses *square hashing* to generate multiple positions for each edge to improve the space utilization of the matrix.

Khan *et al.* [18] propose a 3-D structure called gMatrix, which consists of multiple compressive adjacency matrices with different hash functions. The gMatrix uses reversible hash functions to locate buckets, which can restore the elements based on the position but cause more hash collisions.

All in all, existing graph stream summarization structures are unable to store the temporal information and thus fail to answer a temporal range query. An exact graph stream storage structure [19] provides precise and diverse queries but leads to unacceptable memory cost. In addition to graph stream storage, graph stream processing has also attracted much research interest. Specific graph stream processing includes subgraph matching [20]–[22], triangle counting [23]–[26], event detection [27], [28], maximum matching [29], regular path query [2], and reachability query [30]–[32]. General graph stream processing also attracts recent efforts [17], [33].

In traditional temporal and spatial databases, classical data structures such as the interval tree [12], [13] and the segment tree [14], [15] are widely used for coping with range queries. Given a predefined set of intervals, the interval tree [12], [13] recursively builds a binary tree based on the median of the intervals' endpoints, where each node stores one interval of the set and corresponding data. The right (resp. left) subtree stores the intervals completely to the right (resp. left) of the median. The interval tree can only query the data of the interval set.

The segment tree [14], [15] can support arbitrary range query within a predefined boundary by aggregating the data of several intervals in a top-down style. Given a range boundary, the segment tree builds a static balanced binary tree in a bottom-up manner, where the interval of a non-leaf node is the union of the intervals of its two children while the interval of the root is the boundary. During the insert process, it inserts the data of each point to all the nodes where the intervals contain the point. However, the segment tree cannot address temporal range query over a graph stream which is infinite in time dimension. First, the static construction of the segment tree cannot be applied to a dynamic graph stream. Thus, it cannot guarantee the data integrity at every level of the tree in a one-pass manner, resulting in unacceptable query latency. Second, each node in the segment tree stores the start and the end of a time interval. Such a space cost is unacceptable for a succinct graph stream summarization structure. Third, dynamically building the segment tree requires additional $\log(n)$ computation time when a new interval is inserted, where $n$ is the number of nodes in the tree. New intervals will be continuously inserted to the segment tree as a graph stream evolves. This leads to low throughput. Differently, Horae maintains a matrix for the data of each granularity rather than the data of each interval, and uses an efficient binary range decomposition scheme instead of a top-down style during query processing.

Range division with a binary prefix has been used in other designs [34]–[39]. One Permutation Hashing [34], HyperLogLog [35], and Partitioned Learned Bloom Filter [36] leverage it to reduce the computation overhead, improve the estimation accuracy, and reduce the space overhead as well as the false positive rate, respectively. However, they address different problems other than range queries. Count-Min sketch [37], Persistent Bloom Filter [38], and Rosetta [39] decompose a large range query to multiple binary range queries. Horae is different in three aspects. First, Horae addresses a novel and different problem, *i.e.*, temporal range query over graph streams. Second, Horae proposes the binary range decomposition to perform range queries. Third, Horae can further select and store the subset of all the sub-ranges to achieve a good trade-off between memory cost and query performance.

## III. Problem Statement

**Graph stream**. A graph stream is an evolving graph formed by a timing sequence of elements through a continuous stream. Each element is denoted by ($s_i$, $d_i$, $w_i$, $t_i$) ($i \geq 0$), indicating that the directed edge $s_i \rightarrow d_i$ with a weight value $w_i$ is generated at time $t_i$. For simplicity, we model a discrete graph stream with a time granularity $gl$. In practice, the time granularity can be one minute, one day, and *etc*. It determines the minimum granularity of the temporal range queries in real applications. Accordingly, the continuous time is divided into a sequence of time slices $\{T_0, T_1, \cdots, T_u, \cdots\}$, where $T_i$ represents the $(i+1)^{th}$ time slice of length $gl$ and $T_u$ is the current time slice which increases over time. For a temporal range query, two time slices $T_b$ and $T_e$ ($T_0 \leq T_b \leq T_e \leq T_u$) define a query range $[T_b, T_e]$. For each element ($s_i$, $d_i$, $w_i$, $t_i$) in the graph stream, the edge $s_i \rightarrow d_i$ is produced in one time slice. We use $\gamma(t_i)$ to denote the time slice which time $t_i$ belongs to, and it can be computed by the Eq. (1),

$$\gamma(t_i) = \lfloor t_i / gl \rfloor \tag{1}$$

TABLE I
AN EXAMPLE OF GRAPH STREAM

| The elements of a graph stream | | The time slice |
|---|---|---|
| $(a, b, 1, t_0)$ | $(a, d, 1, t_1)$ | $\gamma(t_0), \gamma(t_1) = T_0$ |
| $(a, b, 1, t_2)$ | $(a, c, 1, t_3)$ | $\gamma(t_2), \gamma(t_3) = T_1$ |
| $(c, d, 1, t_4)$ | $(c, d, 1, t_5)$ | $\gamma(t_4), \gamma(t_5) = T_2$ |
| $(a, d, 1, t_6)$ | $(b, c, 1, t_7)$ | $\gamma(t_6), \gamma(t_7) = T_3$ |

**Example**. Table I shows an example of a graph stream which contains eight elements from time $t_0$ to $t_7$. In the table, two elements in each row are located in the same time slice. Figure 1 shows the time-evolving graph stream. Specially, the same edge may appear multiple times at different moments, *e.g.*, $(a, b, 1, t_0)$ and $(a, b, 1, t_2)$.

**Temporal query primitives**. Here, we define two important query primitives:

- **Temporal edge query**. Given an edge $s \rightarrow d$ and a time range $[T_b, T_e]$, the query returns the aggregated weight of $s \rightarrow d$ within $[T_b, T_e]$. This query covers the existence query, which tests if an edge has appeared within a time range. If the result is zero, $s \rightarrow d$ definitely never appeared within $[T_b, T_e]$. Otherwise, it appeared within $[T_b, T_e]$ with a high probability.
- **Temporal node query**. Given a source (resp. destination) node $v$ and a time range $[T_b, T_e]$, the query returns the aggregated weight of all outgoing (resp. incoming) edges of $v$ within $[T_b, T_e]$.

With the temporal query primitives, we can obtain edge and node information within an arbitrary time range directly. We can also combine these queries to execute more complicated queries, such as temporal path reachability query and temporal subgraph aggregation query. The temporal path reachability query tests whether there is a directed path from one node to another within a time range, and the temporal subgraph aggregation query answers the aggregated weight of a subgraph within a given time range.

Based on the existing summarization structures (*e.g.*, TCM), a straightforward strawman design to support temporal query is to concatenate an edge and its time slice for insertion. Specifically, an element $(s_i, d_i, w_i, t_i)$ is inserted into structures based on $h(s_i \mid \gamma(t_i))$ and $h(d_i \mid \gamma(t_i))$ instead of $h(s_i)$ and $h(d_i)$, where $s_i \mid \gamma(t_i)$ is a concatenation of $s_i$ and $\gamma(t_i)$ while $d_i \mid \gamma(t_i)$ is a concatenation of $d_i$ and $\gamma(t_i)$. Accordingly, we can answer a temporal query "did the edge $x \rightarrow y$ appear at time slice $T$" by checking $M[h(x \mid T)][h(y \mid T)]$.

To evaluate a temporal query with range $[T_b, T_e]$, we need to decompose the range into a set of time slices $\{T_b, T_{b+1}, ..., T_e\}$, and for each time slice we perform a separate query,

$$Q([T_b, T_e]) = Q([T_b]) + Q([T_{b+1}]) + ... + Q([T_e]) \quad (2)$$

The total query processing time increases linearly with the length of the time range. In practice, when the query range becomes large, the query latency can become unacceptable.

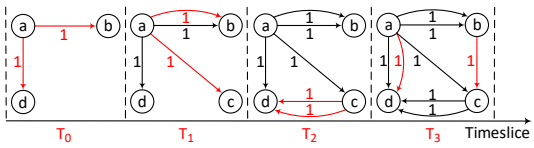

Fig. 1. The formative time-evolving graph

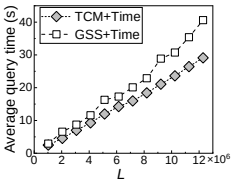

Fig. 2. Average query time of temporal edge queries

In Fig. 2, we examine the performance of the strawmen atop the state-of-the-art TCM and GSS structures using experiments. We use the graph stream formed by the networkflow dataset [40], which contains more than four billion edges with two million nodes. We evaluate temporal edge queries of different range lengths. The result shows that the average query time of existing designs is prohibitively high.

## IV. DESIGN OF HORAE

Based on the above analysis, we propose Horae, a novel time prefix embedded multi-layer summarization structure. We further design an efficient Binary Range Decomposition algorithm. Table II lists the notations in this design.

### A. Overview

The design of Horae is based on the insight that an arbitrary temporal range of length $L$ can be decomposed into at most $2 \log L$ special sub-ranges with two features: 1) all the time slices in each such sub-range have the common binary code prefix; and 2) the prefixes of different such sub-ranges have different sizes. For a simple example, we have $[T_0, T_5] = [T_0, T_3](000, 011) + [T_4, T_5](100, 101)$. The common binary code prefix of all the time slices in the sub-range $[T_0, T_3]$ $(T_0, T_1, T_2, T_3$, *i.e.*, $T_{000}, T_{001}, T_{010}, T_{011})$ is '0', while the common binary code prefix of all the time slices in the sub-range $[T_4, T_5]$ $(T_4, T_5$, *i.e.*, $T_{100}, T_{101})$ is '10'. Formally, we define the ***prefix size of a sub-range*** as the number of binary digits in a common prefix for a sub-range. For example, the prefix size of the sub-range $[T_0, T_3]$ is one, while that of $[T_4, T_5]$ is two.

The Horae structure contains $l = \lceil \log_2(T_u + 1) \rceil + 1$ layers. It starts with a single layer initially and creates one new layer whenever the current time slice of the graph stream increases to a larger power of two $(T_u = 2^i, i \geq 0)$. Horae arranges the layers based on different prefix sizes. Each layer aggregates

TABLE II
NOTATIONS IN HORAE

| Notations | Descriptions |
|---|---|
| $L$ | the length of query range |
| $T_u$ | the current time slice of a graph stream |
| $\gamma(t)$ | the time slice which time $t$ belongs to |
| $\|E\|$ | the number of elements in the graph stream |
| $l$ | the number of layers in current Horae structure |
| $g_p$ | the granularity of the $p^{th}$ layer |
| $M_p$ | the matrix in the $p^{th}$ layer |
| $B_p$ | the adjacency array in the $p^{th}$ layer |
| $W_p^q$ | the window with prefix $q$ in the $p^{th}$ layer |
| $v^q$ | the concatenation of node $v$ and the prefix $q$ |
| $\xi_{v^q}$ | the fingerprint of $v^q$ |
| $A_{v^q}$ | the row/column address of $v^q$ |
| $\lambda$ | the width/depth of matrices in all layers |
| $b$ | the number of entries in each bucket of matrices |
| $F$ | the size of fingerprint |

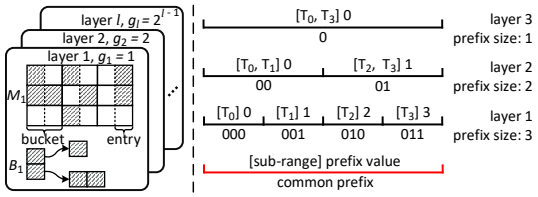


Fig. 3. The structure of Horae with $T_u = T_3$ and $l$ = 3

the complete graph stream data by a corresponding prefix size. Figure 3 illustrates an example, where the right part shows a snapshot with $T_u = T_3$ and the number of layers is three ($l = 3$). In the snapshot, the $1^{st}$ layer contains four sub-ranges $\{[T_0]$ (000), $[T_1]$ (001), ..., $[T_3]$ (011)$\}$, all with the prefix size of three; the $2^{nd}$ layer contains two sub-ranges $\{[T_0, T_1]$ (000, 001), $[T_2, T_3]$ (010, 011)$\}$, both with the prefix size of two; the $3^{rd}$ layer contains one sub-range $\{[T_0, T_3]$ (000, 011)$\}$ with the prefix size of one. Figure 4 illustrates the layout of each layer. Formally, the $p^{th}$ layer aggregates the graph stream data by the sub-ranges $\{[0, 2^{p-1}-1], [2^{p-1}, 2 \cdot 2^{p-1}-1], ...\}$ with the same prefix size of $l - p + 1$. All the sub-ranges of the $p^{th}$ layer have the same range length $2^{p-1}$. We define the same range length of each sub-range in the $p^{th}$ layer as the *granularity* of the $p^{th}$ layer. In a word, the $p^{th}$ layer of Horae represents a graph stream summarization of granularity $2^{p-1}$.

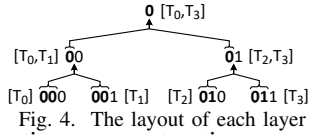


Fig. 4. The layout of each layer

During the continuous construction process, Horae inserts each generated element ($s_i$, $d_i$, $w_i$, $t_i$) into all the layers of the current structure. Specifically, to aggregate data by the sub-ranges, Horae concatenates the edge and the prefix of $\gamma(t_i)$ of size $l - p + 1$ for inserting the edge information into the $p^{th}$ layer. As $T_u$ grows, Horae creates a new layer logarithmically over time whenever $T_u = 2^i$ ($i \geq 0$). Figure 5 shows that Horae creates the $4^{th}$ layer for extension when $T_u$ grows to $T_4$. When $l$ increases by one, the prefix size of each layer $l - p + 1$ needs to increase by one bit by adding a bit of '0' in front of the first bit of each common prefix, specifically. However, the prefix value remains unchanged. For example, the common prefixes of $[T_2, T_3]$ are '01' and '001' when $l = 3$ and $l = 4$, respectively, while the prefix values of '01' and '001' are both '1'. Therefore, we concatenate each edge and the prefix value for insertion. At the same time, the prefix values of the first sub-ranges of different layers are the same under this design, which is essential for Horae's extension. During extension, Horae copies the passed data to guarantee the data integrity of each layer in the one-pass graph stream. For example, when $l$ grows from three to four, the $4^{th}$ layer copies the data of $[T_0, T_3]$ from the $3^{rd}$ layer.

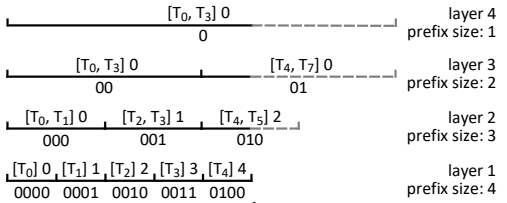


Fig. 5. Horae creates the $4^{th}$ layer when $T_u$ grows to $T_4$

For a temporal edge/node query, Horae first decomposes it into multiple sub-range queries of different layers by the binary range decomposition algorithm. It then concatenates the edge/node and the corresponding prefix value to perform query for each sub-range in the corresponding layer sequentially. The binary range decomposition algorithm achieves a logarithmic scale query processing time (we will present the detail of the algorithm in Section IV-B). For example, given a temporal edge/node query with range $[T_0, T_4]$, Horae first decomposes the range $[T_0, T_4]$ to two sub-ranges $[T_0, T_3]$ (prefix '0') and $[T_4]$ (prefix '4'). Then it concatenates the edge/node and the prefixes (*i.e.*, '0' and '4') to execute the sub-range queries in the $3^{rd}$ and $1^{st}$ layers, respectively.

In each layer, Horae contains a $\lambda \times \lambda$ matrix and an adjacency array to summarize the graph stream with a corresponding granularity. Each bucket of the matrix contains $b$ entries, and each entry maintains a fingerprint pair as well as a weight value. The adjacency array consists of a node array and multiple edge arrays. The adjacency array employs a VoV (*vector of vectors*) format [41], which is efficient in structure extension and data access. We use $M_p$ and $B_p$ to denote the matrix and the adjacency array in the $p^{th}$ layer.

For convenience, we call a sub-range of each layer a *window*. We use the notation $W_p^q$ to denote the window with prefix $q$ in the $p^{th}$ layer, and more generally $W_p^q = [q \cdot 2^{p-1}, (q+1) \cdot 2^{p-1} - 1]$. For example, we have $W_2^2 = [T_4, T_5]$. For a window $[T_i, T_j]$ of length $N$, $[T_i, T_j] = W_{\log_2 N+1}^{\lfloor T_i/N \rfloor} = W_{\log_2 N+1}^{\lfloor T_j/N \rfloor}$. The windows of the $p^{th}$ and $(p+1)^{th}$ layers have the following relationship: the two adjacent windows can be merged into a larger window of the upper layer. Formally, $W_p^{2i} + W_p^{2i+1} = W_{p+1}^i$ ($i \geq 0$), where $W_p^{2i}$ and $W_p^{2i+1}$ are called *sibling windows*.

### B. Operations of Horae

**Insert**. Initially, Horae has only one layer with a single matrix $M_1$ and an empty adjacency array $B_1$. When an element in $T_0$ arrives, Horae first inserts it into the bucket of $M_1$. If its corresponding bucket is full, Horae inserts it into $B_1$. For each element ($s_i$, $d_i$, $w_i$, $t_i$) of $T_0$, $\gamma(t_i) = T_0$, and the window that $T_0$ belongs to is $W_1^0$. Horae concatenates the edge and the prefix '0' to perform the insert operation for aggregating the data within $W_1^0$. Specifically, Horae combines the $F$-bit suffixes of the hash values $h'(s_i \mid 0)$ and $h'(d_i \mid 0)$ to form the fingerprint pair of $s_i \rightarrow d_i$ within $W_1^0$ and uses the remaining bit vectors of $h'(s_i \mid 0)$ and $h'(d_i \mid 0)$ to perform the modulo operation on $\lambda$ to obtain the row address and the column address, respectively. Here, '|' represents the operation of concatenation. For convenience, we use the notations $v^q$, $\xi_{v^q}$, and $A_{v^q}$ to respectively represent the concatenation of node $v$ and the prefix $q$, the fingerprint of $v^q$, and the address of $v^q$. Formally, assuming that the hash value of each node contains $\alpha$ bits, we can obtain $\xi_{v^q}$ and $A_{v^q}$ as follows.

$$\begin{aligned} \xi_{v^q} &= (h'(v^q) << (\alpha - F) >> (\alpha - F)) \\ A_{v^q} &= (h'(v^q) >> F) \% \lambda \end{aligned} \tag{3}$$

Consider an example with $\lambda = 4$, $F = 3$, and $\alpha = 5$. Supposing $h'(s_i^0) = 24$ (11000) and $h'(d_i^0) = 7$ (00111), the

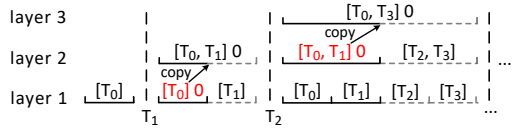


Fig. 6. Extension of Horae

fingerprint pair ($\xi_{s_i^0}$, $\xi_{d_i^0}$) is (000, 111), the row address $A_{s_i^0}$ is 3%4 = 3, and the column address $A_{d_i^0}$ is 0%4 = 0.

Based on the addresses, Horae locates to a bucket of $M_1$, *i.e.*, $M_1[A_{s_i^0}][A_{d_i^0}]$. If all the entries of the bucket are empty, Horae inserts ($\xi_{s_i^0}$, $\xi_{d_i^0}$) and $w_i$ into one entry. Otherwise, Horae checks all the stored fingerprint pairs. If any matched, Horae adds $w_i$ to the weight value stored; otherwise, Horae inserts the element into one empty entry. If all the entries are full and no entries store ($\xi_{s_i^0}$, $\xi_{d_i^0}$), Horae inserts ($[h'(s_i^0), h'(d_i^0)]$, $w_i$) into $B_1$. Horae first finds $h'(s_i^0)$ in the node array. If found, Horae searches $h'(d_i^0)$ in the corresponding edge array. If $h'(d_i^0)$ is in the array, Horae adds $w_i$ to the weight stored in the unit. Otherwise, Horae appends $h'(d_i^0)$ and $w_i$ in the tail of the edge array. If $h'(s_i^0)$ is not in the node array, Horae appends $h'(s_i^0)$ in the tail of the node array and inserts $h'(d_i^0)$ and $w_i$ into the head of the corresponding edge array.

Algorithm 1 describes the insert operation. Horae needs to insert each incoming element into all the current layers. For an element ($s$, $d$, $w$, $t$), the prefix of the window that $\gamma(t)$ belongs to in the $p^{th}$ layer is $\lfloor \gamma(t)/2^{p-1} \rfloor$. In the $p^{th}$ layer, Horae first inserts ($\xi_{s^{\lfloor \gamma(t)/g_p \rfloor}}$, $\xi_{d^{\lfloor \gamma(t)/g_p \rfloor}}$) and $w$ into a bucket of $M_p$ (line 6). If all the entries in the corresponding bucket are full, Horae inserts the element into $B_p$ (line 8). For simplicity, the insert operation in the $p^{th}$ layer performs as inserting ($s^{\lfloor \gamma(t)/2^{p-1} \rfloor}$, $d^{\lfloor \gamma(t)/2^{p-1} \rfloor}$, $w$) into the $p^{th}$ layer.

**Extend**. Figure 6 illustrates an example of Horae's extension. After inserting the elements of $T_0$, the elements of $T_1$ appear. Horae extends the number of layers to two, *i.e.*, it constructs $M_2$ and $B_2$ to the second layer. The size of $M_2$ is the same as that of $M_1$. Note that the elements of $T_0$ have passed, and we can only process the graph stream in one pass. However, the second layer has no data of $T_0$. To guarantee the data integrity of each layer, Horae copies the passed data during extension. Specifically, $M_2$ copies all the data in $M_1$ while $B_2$ copies all the data in $B_1$ to ensure that the elements of $T_0$ are also stored in the $2^{nd}$ layer. With our design, the prefixes of $[T_0]$ and $[T_0, T_1]$ are both '0', and the sizes of $M_1$ and $M_2$ are the same, indicating the position and the fingerprint pair of each element of $T_0$ are the same in the $1^{st}$ and $2^{nd}$ layers. This thus ensures the correctness of the data replication. Since the matrix is continuous in memory and the size of the adjacency array is small (Fig. 32), the data replication cost is slight.

**Algorithm 1: Insert** ($s$, $d$, $w$, $t$)

1 **if** $\gamma(t) == 2^{l-1}$ **then**
2  EXTEND(l);
3  $l = l + 1$;
4 **for** $p = 1$ *to* $l$ **do**
5  $g_p = 2^{p-1}$;
6  insert ($\xi_{s^{\lfloor \gamma(t)/g_p \rfloor}}$, $\xi_{d^{\lfloor \gamma(t)/g_p \rfloor}}$) and $w$ into $M_p$;
7  **if** *insert fails* **then**
8  insert ($[h'(s^{\lfloor \gamma(t)/g_p \rfloor}), h'(d^{\lfloor \gamma(t)/g_p \rfloor})]$, $w$) into $B_p$;
9 **return** *true*.

**Algorithm 2:** Extend ($l$)

1 construct a new matrix $M_{l+1}$ and a new adjacency array $B_{l+1}$;
2 $M_{l+1} = M_l$, $B_{l+1} = B_l$; // copy data
3 **return** *true*.

After completing the extend operation, Horae inserts the elements of $T_1$ into each layer. For each element ($s_i$, $d_i$, $w_i$, $t_i$) of $T_1$, Horae inserts ($s_i^1$, $d_i^1$, $w_i$) into the $1^{st}$ layer and inserts ($s_i^0$, $d_i^0$, $w_i$) into the $2^{nd}$ layer. Similarly, when the elements of $T_2$ arrive, Horae creates the $3^{rd}$ layer, which contains $M_3$ and $B_3$. The matrix and the adjacency array of the $3^{rd}$ layer, *i.e.*, $M_3$ and $B_3$, copy all the data in $M_2$ and $B_2$, respectively. In this way, the data of $[T_0, T_1]$ is also stored in the $3^{rd}$ layer. Then Horae inserts the elements of $T_2$ into all the three layers. More generally, Horae only performs the extend operation when the first element of $T_i$ ($i = 2^{l-1}$) arrives. Algorithm 2 describes the extend operation. Specifically, Horae constructs a new matrix $M_{l+1}$ and a new adjacency array $B_{l+1}$ on the $(l+1)^{th}$ layer. Then, $M_{l+1}$ and $B_{l+1}$ respectively copy all the data stored in $M_l$ and $B_l$ to guarantee the data integrity of each layer.

**Temporal range query**. Basically, in Horae, we can query the data of a window directly in each layer (we call it *window query* for simplicity). We further design the BRD algorithm to quickly decompose an arbitrary time range query to multiple window queries of different layers. For any time range of length $L$, we can find an $m$ which satisfies $2^m \leq L < 2^{m+1}$ (*i.e.*, $m = \lfloor \log_2 L \rfloor$). The granularity of the $(m+1)^{th}$ layer is $2^m$. BRD is based on the observation: if the time range aligns with one side of a window in the $(m+1)^{th}$ layer, decomposing the range equals decomposing its length, *i.e.*, converting $L$ to binary form.

Given an arbitrary query range $[T_b, T_e]$ with the length $L = T_e - T_b + 1$, we formalize the decomposition result as follows: $DE([T_b, T_e]) = W_{p_1}^{q_1} + W_{p_2}^{q_2} + ... + W_{p_f}^{q_f}$, where $W_{p_i}^{q_i}$ and $W_{p_j}^{q_j}$ cannot be sibling windows for any two $i, j \in [1, f]$ and $i \neq j$. Here, we identify an important property of the window: *for a time range $[T_i, T_j]$ of length $N$ ($N = 2^k$ ($k \geq 0$)), it is a window if and only if $(T_j + 1)$ % $N$ = 0*. The property is easy to obtain from the general formula of a window.

For an arbitrary time range, it may align perfectly with a window, align with one side of a window, or not align with any window in the $(m+1)^{th}$ layer. Accordingly, we summarize the following three cases and apply different strategies. Figures 7-9 show examples of the three cases ($4 \leq L < 8$ and $m = 2$).

**Case 1: Perfect alignment**. The time range aligns one window $[k \cdot 2^m, (k+1) \cdot 2^m - 1]$ ($k \geq 0$) perfectly. Obviously, $L = 2^m$ and $[T_b, T_e] = W_{m+1}^{\lfloor T_b / L \rfloor}$. In the example of Fig. 7, we can obtain $DE([T_4, T_7]) = W_3^1$.

**Case 2: Half alignment**. The time range aligns with the left or right side of a window in the $(m+1)^{th}$ layer. Specifically,

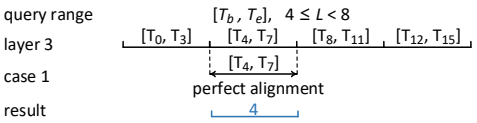


Fig. 7. Perfect alignment

the two situations respectively represent that $T_b = k \cdot 2^m$ and $T_e = (k+1) \cdot 2^m - 1$. We use *left alignment* and *right alignment* to denote the two situations. In both situations, the time range's decomposition is equivalent to the decomposition of its length $L$. We can easily transform $L$ into the addition of multiple second powers by the shift operation. Suppose $L = 2^{x_1} + 2^{x_2} + ... + 2^{x_n}$, where $x_i \ge 0$ and $x_i > x_{i+1}$. Apparently, $2^m < L < 2^{m+1}$ and $x_1 = m$. For left alignment,

$$\begin{aligned}[T_b, T_e] &= [T_b, T_b + 2^{x_1} - 1] + [T_b + 2^{x_1}, T_b + 2^{x_1} + 2^{x_2} - 1] \\ &+ ... + [T_b + 2^{x_1} + ... + 2^{x_{n-1}}, T_e] \\ &= W^{\lfloor \frac{T_b}{2^{x_1}} \rfloor}_{x_1+1} + W^{\lfloor \frac{T_b + 2^{x_1}}{2^{x_2}} \rfloor}_{x_2+1} + ... + W^{\lfloor \frac{T_b + 2^{x_1} + ... + 2^{x_{n-1}}}{2^{x_n}} \rfloor}_{x_n+1}\end{aligned} \tag{4}$$

In the example of Fig. 8, $[T_0, T_5] = [T_0, T_3] + [T_4, T_5]$ as $6 = 2^2 + 2^1$. Hence, $DE([T_0, T_5]) = W^0_3 + W^2_2$. Similarly, for right alignment we have Eq. (5),

$$\begin{aligned}[T_b, T_e] &= [T_e - 2^{x_1} + 1, T_e] + [T_e - 2^{x_1} - 2^{x_2} + 1, T_e - 2^{x_1}] \\ &+ ... + [T_e - 2^{x_1} - ... - 2^{x_n} + 1, T_e - 2^{x_1} - ... - 2^{x_{n-1}}] \\ &= W^{\lfloor \frac{T_e}{2^{x_1}} \rfloor}_{x_1+1} + W^{\lfloor \frac{T_e - 2^{x_1}}{2^{x_2}} \rfloor}_{x_2+1} + ... + W^{\lfloor \frac{T_e - 2^{x_1} - ... - 2^{x_{n-1}}}{2^{x_n}} \rfloor}_{x_n+1}\end{aligned} \tag{5}$$

In Eqs. (4) and (5), we have $T_b = k \cdot 2^m$ ($k \ge 0$) and $T_e = (k+1) \cdot 2^m - 1$ ($k \ge 0$), respectively. It is clear that each time range obtained by decomposition is a window. In addition, these windows are with different sizes, revealing the correctness of the decomposition result.

Algorithm 3 describes the decomposition process of half alignment. Horae leverages the shift operation to obtain the value of each bit and records the size of the second power only when it is '1' (lines 4-6). For a window query of $W^q_p$, Horae needs to know its layer and prefix, *i.e.*, $p$ and $q$. Thus, Horae converts the result windows into a series of $(p, q)$ pairs.

**Case 3: No alignment**. The range $[T_b, T_e]$ aligns with neither of the left and right sides of any window in the $(m+1)^{th}$ layer. As Fig. 9 shows, the case contains two situations that the time range overlaps two and three adjacent windows, respectively. We call the head of each window a *point*, *i.e.,* $q \cdot 2^{p-1}$ ($q \ge 0$, $p \ge 1$). In both situations, we first find the largest point which is not greater than $T_e$ in the layer (we call it the *break point*, denoted as $bp$). The break point can be calculated by $\lfloor T_e / 2^m \rfloor \cdot 2^m$ ($m = \lfloor \log_2 L \rfloor$). Then we divide $[T_b, T_e]$ into $[T_b, bp-1]$ and $[bp, T_e]$. As Fig. 9 shows, the two time ranges after dividing are smaller than $2^{m+1}$ in both situations, and they are right alignment and left alignment, respectively. Even if $[T_b, bp-1]$ or $[bp, T_e]$ is smaller than $2^m$, it is still half alignment. Finally, we calculate their lengths and decompose them by Eqs. (5) and (4).

Algorithm 4 presents BRD in detail. Given a query range, Horae judges which case it belongs to. If it is a perfect

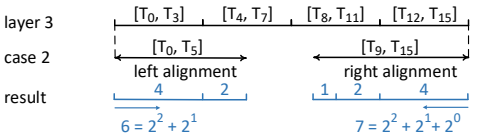


Fig. 8. Half alignment

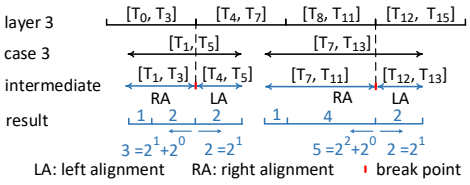


Fig. 9. No alignment

alignment, the range is a window. If it is a half alignment, Horae invokes Algorithm 3. If no alignment, Horae divides the range into two time ranges which are both half alignment and invokes Algorithm 3 twice. For a temporal range query, Horae decomposes the range to multiple windows by the BRD algorithm and then performs the window queries in the corresponding layers. Horae aggregates these intermediates and returns the result.

The temporal query primitives contain the temporal edge and node queries, which are captured by multiple window edge queries and window node queries, respectively. We next present the details in window edge and window node queries.

**Window edge query.** Given an edge $s \to d$ and a window $W^q_p$, Horae concatenates the edge and the prefix $q$ to query in the $p^{th}$ layer. Specifically, Horae obtains the fingerprint pair ($\xi_{s^q}$, $\xi_{d^q}$), the row address $A_{s^q}$, and the column address $A_{d^q}$ based on Eq. (3). Then Horae locates to the bucket $M_p[A_{s^q}][A_{d^q}]$ and checks against all the entries of this bucket whether there is a stored fingerprint pair that matches ($\xi_{s^q}$, $\xi_{d^q}$). If any entry matches, the weight stored in this entry is returned as the result. Otherwise, Horae seeks the edge in $B_p$. Horae first finds $h'(s^q)$ in the node array, and then searches $h'(d^q)$ in the corresponding edge array. If Horae finds the edge in $B_p$, the corresponding weight is returned. Otherwise, zero is returned, indicating $s \to d$ does not exist within $W^q_p$.

**Window node query**. Given a source (resp. destination) node $v$ and a window $W^q_p$, Horae needs to check $M_p$ and $B_p$. Horae first calculates the row (resp. column) address $A_{v^q}$ and the fingerprint $\xi_{v^q}$ based on Eq. (3). According to the address, Horae checks the fingerprint pairs against all the entries of the corresponding row (resp. column) in $M_p$. For any entry that the fingerprint of the source (resp. destination) node is

**Algorithm 3: Decomposition** $(T, L, type, layer)$

```
1  window[], S[];
2  j = 0, k = 0;
3  for i = 0 to (layer − 1) do
4      size = ((L >> i)&1) << i;
5      if size > 0 then
6          S[j++] = size; // different second powers
7  for i = j − 1 to 0 do
8      window[k].p = log2 S[i] + 1;// the layer
9      window[k].q = ⌊T/S[i]⌋;// the prefix
10     k++;
11     if type == left then
12         T = T + S[i];
13     else if type == right then
14         T = T − S[i];
15 return window.
```

**Algorithm 4: BRD** $(T_b, T_e)$

```
L = T_e − T_b + 1, layer = ⌊log_2 L⌋ + 1;
g = 2^(layer−1); // granularity
if T_b % g == 0 then
    if (T_e + 1) % g == 0 then // case 1
        window.p = layer; // the layer
        window.q = T_b/g; // the prefix
        return window.
    else // case 2
        return Decompose(T_b, L, left, layer).
if (T_e + 1) % g == 0 then // case 2
    return Decompose(T_e, L, right, layer);
else // case 3
    bp = ⌊T_e/g⌋ · g; // breakpoint
    L1 = bp − T_b, L2 = T_e − bp + 1;
    window1[]=Decompose(bp − 1, L1, right, layer);
    window2[]=Decompose(bp, L2, left, layer);
    return window1 ⋃ window2.
```

$\xi_{v^q}$, Horae adds the corresponding weight to the result. If $v$ is a source node, Horae finds $h'(v^q)$ in the node array of $B_p$ and searches the corresponding edge array. Otherwise, Horae searches all the edge arrays. For any unit whose source (resp. destination) node's hash value is $h'(v^q)$, Horae adds its weight to the result and returns the aggregated result.

By leveraging the BRD algorithm and multi-layer storage, Horae can decompose an arbitrary time range query to multiple window queries of different layers. In the worst case, Horae needs to divide a time range into two smaller ranges, which can be decomposed to $2\log_2 L$ windows at most.

### C. Optimizations

**The compacted Horae**. To reduce the memory cost, we propose the compacted Horae. In Horae, the matrix of each layer stores complete graph stream data and has the same size. In this way, Horae can achieve a logarithmic scale query processing time. To further improve the space efficiency, the compacted Horae selects partial data to store. Figure 10 shows the scheme. The compacted Horae still has $l$ layers. The first layer stores all the window data as before, and the other layers only maintain the windows with odd prefixes, *i.e.*, each layer except the first layer only stores nearly half of the graph stream data. We set the size of $M_p$ $(p > 1)$ to half of that of $M_1$, which reduces the memory cost by half. For convenience, we use *Horae-cpt* to represent the compacted Horae.

Insert of Horae-cpt performs as follows. For each element $e_i$, if it belongs to the window with an even prefix in the $p^{th}$ $(p > 1)$ layer, Horae-cpt does not perform the insert operation in this layer. Otherwise, it inserts the element into the corresponding layer like Horae. Horae-cpt performs the extend operation only when the first element of $T_i$ $(i = 2^l)$ appears. Moreover, Horae-cpt does not perform the data copy operation because the data of the upper layer is not complete.

Horae-cpt also decomposes a time range query into multiple window queries of different layers. The decomposition process consists of two rounds. Given a time range, we first decompose it into multiple windows by performing BRD once. In the second round, we decompose the windows (length $> 1$) with even prefixes. Suppose $[T_s, T_e]$ $(T_e > T_s)$ is a window with an even prefix. We divide it into $[T_s]$ and $[T_{s+1}, T_e]$ and perform BRD on $[T_{s+1}, T_e]$. Note that the result windows of $DE([T_{s+1}, T_e])$ are all prefixed with odd numbers. Figure 10 shows an example. In the first round, $DE([T_0, T_6]) = [T_0, T_3] + [T_4, T_5] + [T_6]$. In the second round, $[T_0, T_3]$ is decomposed into $[T_0]$, $[T_1]$, and $[T_2, T_3]$ while $[T_4, T_5]$ is decomposed into $[T_4]$ and $[T_5]$. All the windows after the first round decomposition may need binary range decomposition again. It is not difficult to see that Horae-cpt can achieve $O(\log^2 L)$ query latency in the worst case. Horae-cpt achieves a good trade-off between memory cost and query performance.

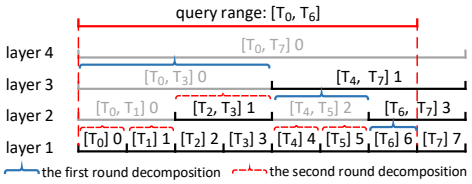


Fig. 10. The storage scheme and query process of the compacted Horae

**Parallel insert**. During the insert operation, Horae needs to insert each element into each layer. A sequential insert operation can result in the decrease of throughput as the number of layers increases. Based on the observation that all the layers of Horae are independent during the insert operation, we exploit parallelism by assigning each layer a thread to perform insert independently. When Horae creates a new layer $l+1$, only the thread of layer $l$ can execute the extend operation to ensure that the insert operation of layer $l$ is performed after the extension. Then, we add a new thread and assign it to the $(l+1)^{th}$ layer. The parallelism between layers requires no locks and greatly improves the throughput.

**Multiple optional positions**. In each layer, it is possible to map multiple different edges to the same bucket of the matrix if there is only one candidate position per edge. Such mapping conflicts lead to a surge of the size of the adjacency arrays, affecting the insert throughput and the query performance. To store as many edges in the matrix as possible, we leverage the linear congruence method [11], [42] to generate multiple candidate positions for each edge. Typically, we can generate four row addresses and four column addresses for the source node as well as the destination node and achieve 16 optional positions. When Horae inserts each edge into one bucket of its 16 positions, Horae needs to store two indexes to indicate the position. Each index represents the number of rounds of the linear congruence iteration.

**Query locality**. In the strategy of multiple positions, randomly generating the addresses can lead to high cost during checking the matrix part for the window edge query and the window destination node query (16 and $4\lambda$ memory accesses in the worst case). To improve query locality, we refer to the alternate ranges proposed in [43]. Here we consider the locality of the four optional positions in each row. We divide each row into multiple chunks with equal sizes which are the same as that of a cache line. Each chunk contains four buckets. We keep every two optional positions adjacent so that they can be placed in the same cache line during one

memory access. Specifically, two column addresses $C_1$ and $C_2$ are randomly generated by the linear congruence, and the remaining two column addresses $C_1'$ and $C_2'$ are generated as follows: $C' = C \oplus (\xi_{d_i^q} \% 4)$, where $\xi_{d_i^q}$ is the fingerprint of the destination node $d_i$ with a prefix $q$.

**Kick out**. We leverage the idea of eviction in cuckoo hashing [44] to improve the space utilization of each matrix. When inserting an element, if all the buckets in the 16 optional positions are occupied, Horae evicts one edge stored in the bucket in the first optional position, which minimizes the computational overhead. According to the power of two choices [45], we evict the edge with the smallest sum of the two indexes among all the entries of the bucket. The edge with a smaller sum has more subsequent optional positions, *i.e.*, the probability of successful insertion into the matrix is larger. If the evicted edge still fails to be inserted into the matrix, Horae kicks one edge stored in the bucket in the second optional position to avoid a looping kick.

## V. Analysis

### A. Overhead Analysis

**Memory overhead**. When constructing Horae, we set the matrix sizes of all the layers to the same value. Specifically, we set the matrix size to a value close to $|E|$ ($\lambda \times \lambda \times b \approx |E|$) though the number of distinct edges in each layer is less than $|E|$. This is to guarantee that each matrix can store all the edges in a dataset. The memory cost of Horae is $O(\log(T_u)|E| + \sum_{i=1}^{l} |B_i|)$, where $|B_i|$ is the size of the adjacency array in the $i^{th}$ layer. Because the proportion of the adjacency array is very small (Fig. 32), the overall memory overhead is $O(\log(T_u)|E|)$. This setting does not mean that Horae can only represent the current graph stream. There are many empty entries in the matrix of each layer to store more data. The memory cost of Horae-cpt is $O((\frac{1}{2}(\log(T_u) - 1) + 1)|E| + \sum_{i=1}^{l} |B_i|)$, still at the scale of $O(\log(T_u)|E|)$.

**Query overhead**. The time complexity of Horae's temporal query is related to the number of window queries by BRD. Let $n$ denote the number of window queries. For an arbitrary temporal query with length $L$, we discuss $n$ in different cases. In case 1, $n = 1$. In case 2, $n \leq \log_2 L$ because the number of bits of $L$ is $\log_2 L$. In case 3, $n \leq 2\log_2 L$. Above all, Horae needs $O(\log L)$ time in the worst case for the temporal query. For a temporal query, Horae-cpt needs to perform two rounds of the BRD algorithm and needs $O(\log^2 L)$ query time in the worst case.

### B. Collision Rate Analysis

In this section, we analyze the node and edge collision rates in Horae. Horae only has one-side error, *i.e.*, the returned result is not less than the exact value.

**Node collision in the $p^{th}$ layer**. For an element $e_i$ = ($s_i$, $d_i$, $w_i$, $t_i$), the source (resp. destination) node collision occurs when there is at least another element ($s_j$, $d_j$, $w_j$, $t_j$) which satisfies the following conditions: (1) $s_i \neq s_j$ (resp. $d_i \neq d_j$) or $\lfloor \gamma(t_i)/2^{p-1} \rfloor \neq \lfloor \gamma(t_j)/2^{p-1} \rfloor$; (2) $h'(s_i^{\lfloor \gamma(t_i)/2^{p-1} \rfloor}) = h'(s_j^{\lfloor \gamma(t_j)/2^{p-1} \rfloor})$ (resp. $h'(d_i^{\lfloor \gamma(t_i)/2^{p-1} \rfloor}) = h'(d_j^{\lfloor \gamma(t_j)/2^{p-1} \rfloor})$. We use $R$ to denote the size of the value range of the hash function and use $Z$ to denote the set of elements with the same source node (resp. destination node) of the same window as $e_i$. For the element that does not belong to $Z$, the probability that it collides with $e_i$ is $\frac{1}{R}$. The probability that $e_i$ suffers from node collision in the $p^{th}$ layer is,

$$Pr_p(node) = 1 - (1 - \frac{1}{R})^{(|E|-|Z|)} \approx 1 - e^{-\frac{|E|-|Z|}{R}} \tag{6}$$

The value range of the hash function is the same for the source nodes and the destination nodes while $R \approx \sqrt{|E|/2} \cdot 2^F$. Given a temporal node query with the range $[T_b, T_e]$ and $DE([T_b, T_e])$ = $W_{p_1}^{q_1} + W_{p_2}^{q_2} + ... + W_{p_f}^{q_f}$, its node collision probability is,

$$\begin{aligned} Pr(node) &= 1 - \prod_{i=1}^{f}(1 - Pr_{p_i}(node)) \approx 1 - \prod_{i=1}^{f}(e^{-\frac{|E|-|Z_i|}{R}}) \\ &\leq 1 - e^{-\frac{f|E|}{R}} \approx 1 - e^{-\frac{f|E|}{\sqrt{|E|/2 \cdot 2^F}}} = 1 - e^{-\frac{f\sqrt{2|E|}}{2^F}} \end{aligned} \tag{7}$$

Because the number of windows obtained by the BRD algorithm, *i.e.*, $f$, is small, if we set the fingerprint's size $F$ to a large value, the node collision probability can be small.

**Edge collision in the $p^{th}$ layer**. For an element $e_i$ = ($s_i$, $d_i$, $w_i$, $t_i$), the edge collision occurs when there is at least another element $e_j$ = ($s_j$, $d_j$, $w_j$, $t_j$) which satisfies the following conditions: (1) $s_i \neq s_j$ or $d_i \neq d_j$ or $\lfloor \gamma(t_i)/2^{p-1} \rfloor \neq \lfloor \gamma(t_j)/2^{p-1} \rfloor$; (2) $h'(s_i^{\lfloor \gamma(t_i)/2^{p-1} \rfloor}) = h'(s_j^{\lfloor \gamma(t_j)/2^{p-1} \rfloor})$ and $h'(d_i^{\lfloor \gamma(t_i)/2^{p-1} \rfloor}) = h'(d_j^{\lfloor \gamma(t_j)/2^{p-1} \rfloor})$. The probability that $e_i$ suffers from edge collision in the $p^{th}$ layer is,

$$\begin{aligned} Pr_p(edge) &= 1 - (1 - \frac{1}{R^2})^{(|E|-D)} \cdot (1 - \frac{1}{R})^D \\ &\approx 1 - e^{-\frac{|E|+(R-1)D}{R^2}} \end{aligned} \tag{8}$$

where $D$ is the number of elements with the same source node or destination node of the same window as $e_i$ in the $p^{th}$ layer. We define these elements as *similar elements*. For each similar element, the probability that it collides with $e_i$ is $\frac{1}{R}$. For other elements, the collision probability is $\frac{1}{R^2}$.

Similarly, given a range $[T_b, T_e]$ and $DE([T_b, T_e])$ = $W_{p_1}^{q_1} + W_{p_2}^{q_2} + ... + W_{p_f}^{q_f}$, the edge collision probability of a temporal edge query is computed by,

$$\begin{aligned} Pr(edge) &= 1 - \prod_{i=1}^{f}(1 - Pr_{p_i}(edge)) \\ &\approx 1 - e^{-\frac{f|E|+(R-1)\sum_{i=1}^{f} D_i}{R^2}} \\ &= 1 - e^{-f(\frac{1}{2^{2F}-1} + \frac{D_{max}}{\sqrt{|E|/2 \cdot 2^F}})} \end{aligned} \tag{9}$$

where $D_{max}$ denotes the maximum value of $D_i$. Since $f$ and $D_{max}$ are small, the edge collision rate is much lower than the node collision rate.

### C. Error Bound

We analyze the error bound and the setting of $F$ as follows.

**Window node query**. The result of a window source node query is denoted as $\hat{w}(x')$, where $x'$ denotes node $x$ with its window prefix. We assume that the exact value is $w(x')$. Given a parameter $\varepsilon$, we set $F = \lceil \log(e/(\varepsilon\sqrt{|E|/2})) \rceil$, then $R = e/\varepsilon$. We use $\|\boldsymbol{w}\|$ to represent the aggregated weights of all the elements.

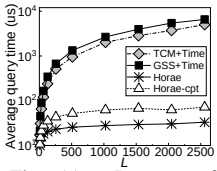


Fig. 11. Latency of temporal edge query on lkml

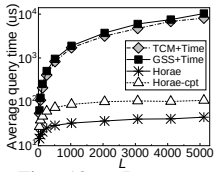


Fig. 12. Latency of temporal edge query on wiki

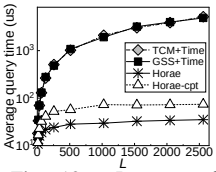


Fig. 13. Latency of temporal edge query on stackoverflow

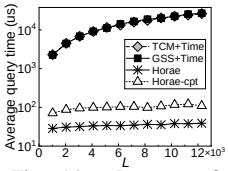


Fig. 14. Latency of temporal edge query on IP flow

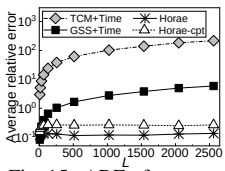


Fig. 15. ARE of temporal edge queries on lkml

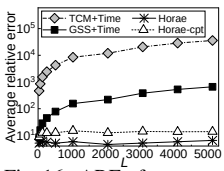


Fig. 16. ARE of temporal edge queries on wiki

**THEOREM 5.1:** The result $\hat{w}(x')$ guarantees $\hat{w}(x') \leq w(x') + \varepsilon \|\boldsymbol{w}\|$ with a probability of at least $1 - e^{-1}$.

**Proof.** Consider two elements ($x$, $y$, $w_1$, $t_1$) and ($s$, $d$, $w_2$, $t_2$). We introduce an indicator $\boldsymbol{I}_{x',s'}$ which is one if there is a source node collision between the two elements and zero otherwise. We have the expectation of $\boldsymbol{I}_{x',s'}$,

$$E(\boldsymbol{I}_{x',s'}) = Pr[(h'(x') = h'(s'))] \leq 1/R = \varepsilon/e \tag{10}$$

where $e$ is the Euler's number. Let $\boldsymbol{X}_{x'}$ denote $\sum_{i=1}^{|E|} \boldsymbol{I}_{x',s'_i} w_i$. By construction, $\hat{w}(x') = w(x') + \boldsymbol{X}_{x'}$. By linear expectation,

$$E(\boldsymbol{X}_{x'}) = E(\sum_{i=1}^{|E|} \boldsymbol{I}_{x',s'_i} w_i) \leq \sum_{i=1}^{|E|} w_i E(\boldsymbol{I}_{x',s'_i}) \leq (\varepsilon/e) \|\boldsymbol{w}\| \tag{11}$$

By applying the Markov inequality, we have,

$$\begin{aligned} &Pr[\hat{w}(x') > w(x') + \varepsilon \|\boldsymbol{w}\|] \\ &= Pr[w(x') + \boldsymbol{X}_{x'} > w(x') + \varepsilon \|\boldsymbol{w}\|] \\ &= Pr[\boldsymbol{X}_{x'} > \varepsilon \|\boldsymbol{w}\|] = Pr[\boldsymbol{X}_{x'} > eE(\boldsymbol{X}_{x'})] < e^{-1} \end{aligned} \tag{12}$$

The window destination node query can be analyzed by a similar discussion, which has the similar guarantee.

**Temporal node query**. We use $\hat{w}_{node}[T_b, T_e]$ to denote the result of a temporal node query of the range $[T_b, T_e]$ with length $L$.

**THEOREM 5.2:** The result $\hat{w}_{node}[T_b, T_e]$ has the guarantee: $\hat{w}_{node}[T_b, T_e] \leq w_{node}[T_b, T_e] + 2\varepsilon \log L \|\boldsymbol{w}\|$ with a probability of at least $1 - e^{-1}$.

**Proof.** For any temporal range node query, it can be decomposed into $2 \log L$ window node queries at most. Applying the inequality of Theorem 5.1 to each window node query, the expectation of the absolute error of $\hat{w}_{node}[T_b, T_e]$ is $2 \log L(\varepsilon/e) \|\boldsymbol{w}\|$. Similarly, applying the Markov inequality, the probability that the absolute error is more than $2\varepsilon \log L \|\boldsymbol{w}\|$ is less than $e^{-1}$.

**Temporal edge query**. We use $\hat{w}(x', y')$ to denote the result of a temporal edge query with a window and use $\hat{w}_{edge}[T_b, T_e]$ to denote the result of a temporal edge query with the range $[T_b, T_e]$, respectively. With the same setting of $F$, we have the following guarantee. We omit the similar proof.

**LEMMA 5.1:** The result $\hat{w}(x', y')$ guarantees $\hat{w}(x', y') \leq w(x', y') + \varepsilon^2/e \|\boldsymbol{w}\|$ with a probability of at least $1 - e^{-1}$.

**LEMMA 5.2:** The result $\hat{w}_{edge}[T_b, T_e]$ guarantees $\hat{w}_{edge}[T_b, T_e] \leq w_{edge}[T_b, T_e] + 2\varepsilon^2/e \log L \|\boldsymbol{w}\|$ with a probability of at least $1 - e^{-1}$.

### D. Data Distribution Analysis

The distribution of incoming data can be uniform or non-uniform over time. In the following, we explain that the data distribution has a slight impact on this design.

Horae includes time and node as input when computing the hash value during construction. More importantly, the data of each layer is stored in one matrix. Under the assumption that $|E|$ does not change regardless of the data distribution, we can see that the edge collision is only affected by the number of similar elements for one element from Eq. (8). However, the collision is determined by the size of fingerprint $F$ and is less affected by the number of similar elements. Hence, the collision rate is close. It is possible that the high-degree nodes appear in the windows with a lot of data under the non-uniform distribution. The row (resp. column) addresses of the edges with the same high-degree source (resp. destination) node within the same window are the same. This may increase the mapping conflicts and the size of the adjacency arrays, thereby affecting the latency of the destination node query and the insert throughput. The experimental results in Table III are in agreement with our analysis.

## VI. Experiment

### A. Experimental Setup

We implement Horae and make the source code publicly available[1]. In the evaluation, we use the state-of-the-art designs TCM [1] and GSS [11] as the baselines. Since neither TCM nor GSS directly supports temporal range queries, we extend both of them with the range decomposition scheme presented in Section III to enable them to support temporal range queries. We use four large-scale real-world datasets and sort each in chronological order to form graph streams.

1) **Lkml [46]**. It is a communication network of the mailing list of Linux kernel, where a node represents a user (email address) and a directed edge denotes a reply. The dataset contains $63,399$ users and $1,096,440$ replies. We set the time granularity $gl$ to one day and obtain $2,922$ time slices.

2) **Wiki [47]**. It is the interaction network of the English Wikipedia, where a node denotes a user and an edge denotes a message from one user to another at a certain time. The dataset contains $2,987,535$ nodes, $24,981,163$ messages, and $5,363$ time slices. Each time slice is one day.

3) **Stackoverflow [48]**. It is an interaction network between users collected from "Stack Overflow" during 2009 to 2016. It contains $2,601,977$ nodes and $63,497,050$ edges. We divide the dataset by day and obtain 2,775 time slices.

4) **IP flow [40]**. This dataset contains partial anonymized passive traffic traces from CAIDA's equinox-Chicago monitor in 2015. A node represents an IP address and an edge denotes a communication. The weight of each edge represents the size of the transmitted data. The collected eleven minutes trace contains $2,121,486$ nodes, $403,436,907$ edges. We set $gl$ to 50 ms and obtain $13,200$ time slices.

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

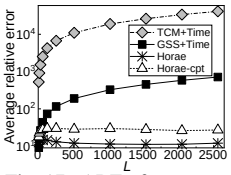


Fig. 17. ARE of temporal edge query on stackoverflow

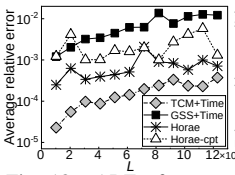


Fig. 18. ARE of temporal edge query on IP flow

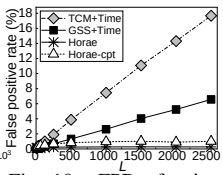


Fig. 19. FPR of existence query on lkml

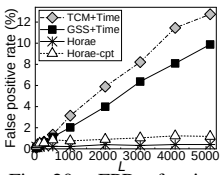


Fig. 20. FPR of existence query on wiki

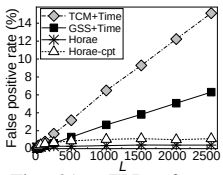


Fig. 21. FPR of existence query on stackoverflow

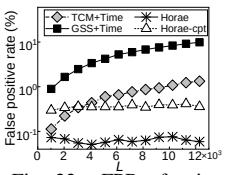


Fig. 22. FPR of existence query on IP flow

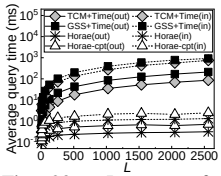


Fig. 23. Latency of temporal node queries on lkml

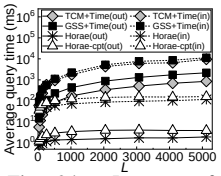


Fig. 24. Latency of temporal node queries on wiki

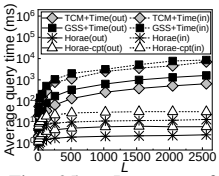


Fig. 25. Latency of temporal node queries on stackoverflow

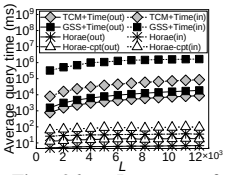


Fig. 26. Latency of temporal node queries on IP flow

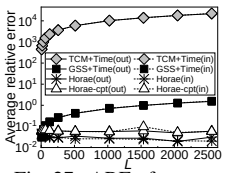


Fig. 27. ARE of temporal node queries on lkml

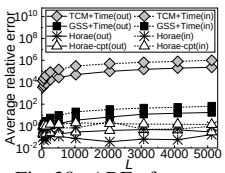


Fig. 28. ARE of temporal node queries on wiki

We conduct the experiments on a machine with a 16-core 2.6 GHz Xeon CPU, 64 GB RAM, and 1 TB HDD. We implement Horae and the baselines with the same memory configuration. In TCM, we use four different hash functions. In Horae and GSS, the number of optional addresses is four for two nodes of each edge. We use $2 \times 2 = 4$ bits to store the two indexes, which indicate the position. The fingerprint size largely determines the collisions. We set $F$ to six and 14 bits in temporal edge and node query experiments, respectively. While keeping the size of each matrix to $O(|E|)$, we set $b$ to two to improve the matrix utilization and set the width/depth of the matrix smaller. In Horae-cpt, we set the size of $M_p$ $(p > 1)$ to half of that of $M_1$.

We conduct six groups of experiments. The first two groups evaluate the performance of two primitive queries. We vary the value of $L$ to see how it influences the query performance. For each value of $L$, we generate a set of temporal edge queries with the cardinality of $100,000$ and a set of temporal node queries with the cardinality of $10,000$. In each query, the edge/node and the query range are randomly generated. The third group evaluates the memory cost and the insert throughput. The fourth group evaluates the optimization strategies. The fifth group compares Horae with the segment tree. The sixth group examines the influence of data distribution.

The metrics include the average query time, the insert throughput, the *average relative error* (ARE), the *false positive rate* (FPR), and the memory cost. The average query time quantifies the average query latency. The insert throughput measures the number of elements inserted per second. The *average relative error* (ARE) quantifies the error between the exact value and the query result. For a query $Q$, supposing $f(Q)$ is the exact value of $Q$ and $\widetilde{f}(Q)$ is the query result, the *relative error* is quantified as $(\widetilde{f}(Q) - f(Q))/f(Q)$. The ARE is the average relative error over all the queries.

If an edge does not exist within the specified time range, we call it a *non-existence query*. For a set of non-existence queries $\vartheta$, the *false positive rate* (FPR) can be calculated by $\text{FPR}(\vartheta) = K/|\vartheta|$, where $K$ is the number of non-existence queries whose results are greater than zero. The memory cost measures the memory consumption of the structure.

### B. Results

**Temporal edge weight queries**. In each query of the test sets, we guarantee that the edge appears at least once within the given time range. Figures 11-14 show the average query time of temporal edge weight queries for the datasets lkml, wiki, stackoverflow, and IP flow, respectively. Horae and Horae-cpt both greatly reduce the average latency of GSS and TCM by over two orders of magnitude.

Figures 15-18 plot the ARE of the temporal edge weight queries with different lengths. Horae reduces the AREs of GSS and TCM by one order of magnitude and two orders of magnitude, respectively. The ARE of Horae-cpt is slightly larger than that of Horae.

**Temporal edge existence queries**. We examine the non-existence queries where the edges do not appear within the corresponding time ranges. Figures 19-22 show the FPR of the non-existence queries for the four datasets. The FPR increases significantly for GSS and TCM as $L$ increases. In contrast, it maintains stable in Horae. All in all, Horae reduces the FPR of GSS by 93.7%-99% and reduces that of TCM by 95.5%-98.5% for the four datasets, respectively, when $L$ is large. The FPR of Horae-cpt is close to that of Horae.

**Temporal node queries**. We use the temporal node-out (resp. node-in) query to represent the temporal source (resp. destination) node query. Figures 23-26 plot the average time of temporal node-in and node-out queries with different lengths. Horae reduces the latency of GSS by nearly three orders of magnitude when the length is considerable while it reduces that of TCM more. The latency of Horae-cpt is slightly larger than that of Horae.

Figures 27-30 illustrate the ARE of the temporal node queries with different lengths for the four datasets. Horae reduces the ARE of GSS by almost two orders of magnitude for both the temporal node-out queries and the temporal node-in queries. The ARE of temporal node query of TCM is high because it lacks a collision avoidance mechanism. The ARE of node query in Horae-cpt is close to that in Horae.

**Memory cost**. In the experiment, we configure Horae, GSS, and TCM with the same amount of memory. The result in Fig. 31 shows that Horae-cpt greatly reduces the memory cost

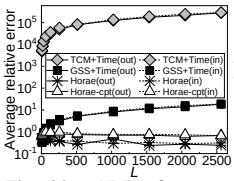


Fig. 29. ARE of temporal node queries on stackoverflow

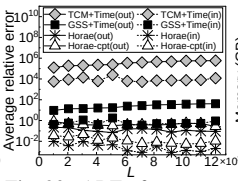


Fig. 30. ARE of temporal node queries on IP flow

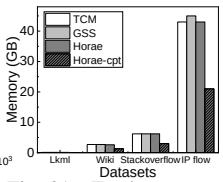


Fig. 31. Total memory cost on the four datasets

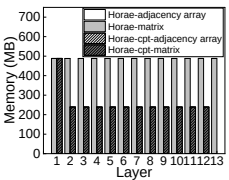


Fig. 32. Memory breakdown of Horae on stackoverflow

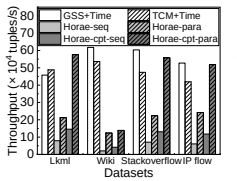


Fig. 33. Insert throughput for four datasets

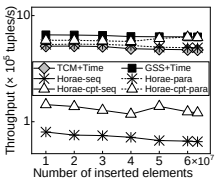


Fig. 34. Insert throughput on stackoverflow

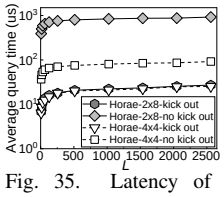


Fig. 35. Latency of temporal edge queries on stackoverflow

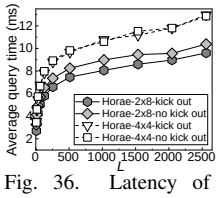


Fig. 36. Latency of temporal node queries on stackoverflow

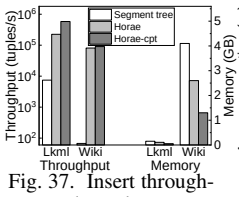


Fig. 37. Insert throughput and total memory cost

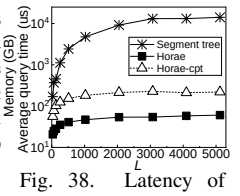


Fig. 38. Latency of temporal edge queries on wiki

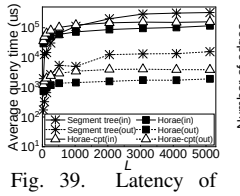


Fig. 39. Latency of temporal node queries on wiki

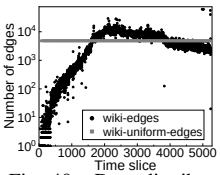


Fig. 40. Data distributions of wiki and wiki-uniform

of Horae by 50%. Figure 32 shows the memory breakdown of each layer of Horae and Horae-cpt on stackoverflow. The proportions of memory cost for the adjacency array are very low and close to zero in both Horae and Horae-cpt.

**Insert throughput.** Figures 33 shows the insert throughput of GSS, TCM, and Horae for the four datasets. Horae-para and Horae-seq denote the parallel insert operation and the sequential insert operation, respectively. Figure 34 shows the average insert throughput of every ten million elements for stackoverflow. The parallel insert operation greatly improves the throughput compared to the sequential one. The throughput of Horae-cpt-para is close to those of GSS and TCM.

**Optimization.** Figures 35-36 illustrate the average latency of the temporal edge query and the node-in query for different optimizations. In the figures, "$4 \times 4$" and "$2 \times 8$" indicate that the 16 positions of each element are generated by the linear congruence and the locality strategies, respectively. The $2 \times 8$-kick-out represents the final version, which is used in the first four groups of experiments. The kick-out strategy reduces the average query time of the temporal edge query and the temporal node-in query by 97% and 51% with "$2 \times 8$", while it reduces those by 71% and 4.3% with "$4 \times 4$". In Fig. 36, the locality strategy performs well in the temporal node-in queries and reduces the query time by 25%.

**Comparison with segment tree**. We implement a dynamic segment tree based on Horae's expansion strategy to support the temporal query over graph streams. Figure 37 plots the memory cost and the insert throughput on the datasets lkml and wiki. Horae reduces the memory cost of the segment tree by 36.6%. The throughput of the segment tree is only 67 tuples per second on wiki, which is unacceptable in practice. Figures 38-39 show the query latency of the temporal edge query and the temporal node query. Horae reduces the query latency of the segment tree by one to two orders of magnitude.

TABLE III
DATA DISTRIBUTION EXPERIMENT

| Metrics \ Datasets | Horae | | Horae-cpt | |
|---|---|---|---|---|
| | wiki | wiki-uniform | wiki | wiki-uniform |
| Throughput ($10^4$ tuples/s) | 10.22 | 10.32 | 12.87 | 18.03 |
| Edge query time (us) | 6.63 | 5.26 | 9.04 | 8.19 |
| Node-out query time (ms) | 0.59 | 0.60 | 1.28 | 1.38 |
| Node-in query time (ms) | 55.93 | 42.96 | 98.99 | 54.11 |
| ARE of edge query | 1.23 | 1.28 | 1.81 | 1.93 |
| ARE of node-out query | 0.39 | 0.41 | 0.66 | 0.69 |
| ARE of node-in query | 0.76 | 0.79 | 1.67 | 1.65 |

**Data distribution**. We use a non-uniform dataset wiki to examine the influence of data distribution on Horae. For comparison, we adjust the timestamps of wiki to generate a uniform dataset, namely wiki-uniform. Figure 40 illustrates the distributions of wiki and wiki-uniform datasets. Table III shows Horae performs similarly on the two datasets, so does Horae-cpt. For wiki-uniform, the latency of node-in query is slightly lower and the throughput is slightly higher. This is more obvious in Horae-cpt, because the upper layers store more data on wiki than wiki-uniform.

## VII. Conclusion and Future Work

In this paper, we propose Horae, a novel graph stream summarization structure for efficient temporal range query. By designing a time prefix embedded multi-layer structure and a binary range decomposition algorithm for temporal range query processing, Horae reduces the query latency to a logarithmic scale while maintaining high accuracy of query results. Both theoretical analysis and comprehensive experimental results demonstrate the superiority of Horae. There are two issues that are worth further investigation, (1) how to extend Horae matrices to store infinite data as they tend to become saturated when a graph stream grows; (2) how to summarize a multi-dimension graph stream, where each element contains multiple edge labels and node labels.

## VIII. Acknowledgements

This research is supported in part by the National Key Research and Development Program of China under grant No.2018YFB1004602, NSFC under grant No. 61972446, Key Research and Development Program of Hubei Province No.2021BEA164, and Huawei Research Project No. TC20210702017. The research by Prof. Bo Li was support in part by RGC RIF grant R6021-20, and RGC GRF grants under the contracts 16207818 and 16209120.

## REFERENCES


[1] 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, pp. 1481–1496.

[2] 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, pp. 1415–1430.

[3] M. S. Hassan, B. Ribeiro, and W. G. Aref, "Sbg-sketch: a self-balanced sketch for labeled-graph stream summarization," in *Proceedings of SSDBM,* Bozen-Bolzano, Italy, July 09-11, 2018, pp. 3:1–3:12.

[4] L. Zervakis, V. Setty, C. Tryfonopoulos, and K. Hose, "Efficient continuous multi-query processing over graph streams," in *Proceedings of EDBT,* Copenhagen, Denmark, March 30 - April 02, 2020, pp. 13–24.

[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] A. C and K. Mahesh, "Graph analytics applied to COVID19 karnataka state dataset," in *Proceedings of ICISS,* Edinburgh, United Kingdom, March 17-19, 2021, pp. 74–80.

[7] B. Tian, B. T. Morris, M. Tang, Y. Liu, Y. Yao, C. Gou, D. Shen, and S. Tang, "Hierarchical and networked vehicle surveillance in ITS: A survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. 18, no. 1, pp. 25–48, 2017.

[8] A. Khan, S. S. Bhowmick, and F. Bonchi, "Summarizing static and dynamic big graphs," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1981–1984, 2017.

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

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

[11] 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, pp. 1118–1129.

[12] H. Edelsbrunner, "Dynamic data structures for orthogonal intersection queries." *Technical Report F59,* Inst. Informationsverarb., Tech. Univ. Graz, Graz, Austria, 1977.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2009.

[14] M. De Berg, M. Van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry*. Springer, 1997.

[15] J. L. Bentley, "Solution to klee's rectangle problems," *Technical report,* CarnegieMellon Univ., Pittsburgh, PA, 1977.

[16] Q. Yong, M. Hajiabadi, V. Srinivasan, and A. Thomo, "Efficient graph summarization using weighted LSH at billion-scale," in *Proceedings of SIGMOD,* China, June 20-25, 2021, pp. 2357–2365.

[17] M. Mariappan, J. Che, and K. Vora, "Dzig: sparsity-aware incremental processing of streaming graphs," in *Proceedings of EuroSys,* United Kingdom, April 26-28, 2021, pp. 83–98.

[18] A. Khan and C. C. Aggarwal, "Query-friendly compression of graph streams," in *Proceedings of ASONAM,* San Francisco, CA, USA, August 18-21, 2016, pp. 130–137.

[19] F. Zhang, L. Zou, L. Zeng, and X. Gou, "Dolha - an efficient and exact data structure for streaming graphs," *World Wide Web*, vol. 23, no. 2, pp. 873–903, 2020.

[20] 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, pp. 411–426.

[21] 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, pp. 1082–1093.

[22] C. Wang and L. Chen, "Continuous subgraph pattern search over graph streams," in *Proceedings of ICDE,* Shanghai, China, March 29 - April 2, 2009, pp. 393–404.

[23] P. Wang, Y. Qi, Y. Sun, X. Zhang, J. Tao, and X. Guan, "Approximately counting triangles in large graph streams including edge duplicates with a fixed memory usage," *Proceedings of the VLDB Endowment*, vol. 11, no. 2, pp. 162–175, 2017.

[24] L. D. Stefani, A. Epasto, M. Riondato, and E. Upfal, "Trièst: Counting local and global triangles in fully-dynamic streams with fixed memory size," in *Proceedings of SIGKDD,* San Francisco, CA, USA, August 13-17, 2016, pp. 825–834.

[25] K. Shin, S. Oh, J. Kim, B. Hooi, and C. Faloutsos, "Fast, accurate and provable triangle counting in fully dynamic graph streams," *ACM Transactions on Knowledge Discovery from Data*, vol. 14, no. 2, pp. 12:1–12:39, 2020.

[26] X. Gou and L. Zou, "Sliding window-based approximate triangle counting over streaming graphs with duplicate edges," in *Proceedings of SIGMOD,* China, June 20-25, 2021, pp. 645–657.

[27] 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.

[28] 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, pp. 1387–1388.

[29] R. Chitnis, G. Cormode, H. Esfandiari, M. Hajiaghayi, A. McGregor, M. Monemizadeh, and S. Vorotnikova, "Kernelization via sampling with applications to finding matchings and related problems in dynamic graph streams," in *Proceedings of SODA,* Arlington, VA, USA, January 10-12, 2016, pp. 1326–1344.

[30] T. Zhang, Y. Gao, L. Chen, W. Guo, S. Pu, B. Zheng, and C. S. Jensen, "Efficient distributed reachability querying of massive temporal graphs," *The VLDB Journal.*, vol. 28, no. 6, pp. 871–896, 2019.

[31] A. D. Zhu, W. Lin, S. Wang, and X. Xiao, "Reachability queries on large dynamic graphs: a total order approach," in *Proceedings of SIGMOD,* Snowbird, UT, USA, June 22-27, 2014, pp. 1323–1334.

[32] X. Chen, K. Wang, X. Lin, W. Zhang, L. Qin, and Y. Zhang, "Efficiently answering reachability and path queries on temporal bipartite graphs," *Proceedings of the VLDB Endowment.*, vol. 14, no. 10, pp. 1845–1858, 2021.

[33] P. Vaziri and K. Vora, "Controlling memory footprint of stateful streaming graph processing," in *Proceedings of ATC,* Virtual Event, 2021, pp. 269–283.

[34] P. Li, A. B. Owen, and C. Zhang, "One permutation hashing," in *Proceedings of NIPS,* Lake Tahoe, Nevada, United States, 2012, pp. 3122–3130.

[35] 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, pp. 127–146.

[36] K. Vaidya, E. Knorr, M. Mitzenmacher, and T. Kraska, "Partitioned learned bloom filters," in *Proceedings of ICLR,* Virtual Event, Austria, May 3-7, 2021.

[37] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[38] Y. Peng, J. Guo, F. Li, W. Qian, and A. Zhou, "Persistent bloom filter: Membership testing for the entire history," in *Proceedings of SIGMOD,* Houston, TX, USA, June 10-15, 2018, pp. 1037–1052.

[39] S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, and S. Idreos, "Rosetta: A robust space-time optimized range filter for key-value stores," in *Proceedings of SIGMOD,* online conference Portland, OR, USA, June 14-19, 2020, pp. 2071–2086.

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

[41] "Apache giraph," http://giraph.apache.org, 2021.

[42] 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.

[43] M. Wang, M. Zhou, S. Shi, and C. Qian, "Vacuum filters: More space-efficient and faster replacement for bloom and cuckoo filters," *Proceedings of the VLDB Endowment*, vol. 13, no. 2, pp. 197–210, 2019.

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

[45] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[46] "Lkml-reply," http://konect.cc/networks/lkml-reply, 2021.

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

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