GeminiSketch: An Accurate and Efficient Sketch for Summarizing Temporal Graph Streams with Rolling-out Elimination

Xuyang Jing[†], Chenhao Zhang[†], Zheng Yan[†], Qingze Jiang[†], Witold Pedrycz[‡], Mingjun Wang[†], and Cong Wang[§]
[†]*School of Cyber Engineering, Xidian University, Xi'an, China*
[‡]*Department of Electrical and Computer Engineering, University of Alberta, Edmonton, Canada*
[§]*Northwestern Polytechnical University, Xi'an, China*

***Abstract*—A temporal graph stream represents a graph stream with dynamic behaviors, where the edges connecting vertices change over time. It can model a wide range of network behaviors, e.g., describing changes of subscribership in a social network, monitoring call duration within a mobile network. Analyzing the temporal graph streams needs to timely eliminate expired edges as time goes on. However, existing graph stream summarization methods struggle to adequately handle the above temporal characteristic, which leads to problems of query efficiency and accuracy. In this paper, we propose GeminiSketch, a novel accurate and efficient sketch for summarizing temporal graph streams with rolling-out elimination. GeminiSketch adaptively partitions edges into two time sublines based on temporal information and stores them independently to reduce summarization error. For temporal graph queries, it can quickly access the storage information from partitions, which addresses the efficiency problem. Furthermore, GeminiSketch adopts a new *Rolling-out* elimination technique to efficiently and accurately remove expired edges from large volumes of edges without fully traversing edges, solving the accuracy problem caused by expired edges. Experimental results demonstrate that GeminiSketch significantly outperforms prior work by supporting various types of temporal graph queries, achieving smaller errors and higher process speed under same experiment settings. The source code of GeminiSketch is available on GitHub.**

***Index Terms*—Temporal graph streams, graph summarization, graph sketch**

I. Introduction

A temporal graph stream refers to an unbounded sequence of data items that represent graph edges with exact temporal information. The temporal information takes the form of a time point or a time interval, denoting the existence or active time of edges. The data items can be summarized to form dynamic graph structures and the time-related graph changes are modeled as edges continuously appear and disappear [1]–[5]. Analyzing temporal graph streams can timely capture the relationship changes of vertices, which plays an important role in many network scenarios, e.g., social relationship mining where the edges can denote the activity that one user follows other users at a particular time [6], [7], network monitoring where the edges can represent communications between hosts during a time interval [8], [9], and financial transaction monitoring where the edges can be a time point [10], [11].

Corresponding author: Zheng Yan (zyan@xidian.edu.cn)

At present, two solutions are widely applied to analyze temporal graph streams. The first is the log-based method, where a log structure is used to record the change events of the edges (activation or deactivation) at different time points or time intervals [12]–[16]. For answering temporal queries, active edges are selected from the log structure to construct the graph. However, this method requires sufficient memory and has a low query speed since the time complexity of traversing edges is linear with the number of change events. The second is the snapshot-based method, where the states of the temporal graph at different time points or time intervals are represented as a time-ordered sequence of snapshots [17]–[19]. Each snapshot is a representation of the graph built by the active edges. Graph queries can be executed on a single snapshot or sequentially across several snapshots. However, this method needs to build the complete graph structure at each time point/interval, even if there are few edge changes between two consecutive snapshots. In summary, the above two methods suffer from the problems of excessive storage consumption and inefficiency, which have not been addressed well so far.

Practically, we face a number of challenges in solving the above problems. The first is that it is difficult to analyze temporal graph streams in a memory-efficient and one-pass manner. The data volumes of the temporal graph streams created by real-world applications are tremendously large. For example, 800 million WeChat users generate over 1 billion daily commercial transactions [20]. Hence, analyzing temporal graph streams should require sublinear memory costs and satisfy the one-pass requirement of large-volume data analysis. Nowadays, graph stream summarization leverages hash functions to downscale graphs and stores graph within a compressed matrix, which requires less storage space and enables efficient graph queries than original graph representations [21]–[23]. There are many efforts in summarizing graph streams [24]–[36]. However, summarization techniques for temporal graph streams have not been well studied so far since it is difficult to solve the challenges of query efficiency and accuracy brought by the temporal characteristic.

The second challenge is that it is hard to distinguish expired edges outside the window without recording exact

arrival orders of edges. In temporal graphs, the importance of edges decays over time. The newly arrived edges are more significant than the expired ones since they reflect current graph structure and further change trends [1]–[3]. For example, in a recommendation system, people may buy a lot of beer during holidays. However, when the holidays are over, we need to recommend work supplies rather than beer. The system should adjust the recommendation algorithms on time by eliminating expired edges in the shopping graph and paying much attention to new ones according to temporal information. We can consider expired edges with some regularity as periodic items and prepare some memory in advance before the next period. To eliminate expired edges, a straightforward method is to record arrival orders of all edges and remove expired edges one by one over time. However, this method requires a large amount of memory. Hence, it is difficult to accurately distinguish expired edges from tremendous edges without the guidance of an explicit edge arrival sequence.

The third challenge is that it is not easy to achieve fast edge traversal to improve the efficiency of various types of temporal graph queries. Rapidly querying temporal graphs poses a challenge as it entails scouring eligible edges from a large number of edges in accordance with the query time. Taking network traffic monitoring as an illustrative example. When answering the query about the number of distinct connections that occur in a particular time interval, analysts have to painstakingly traverse multiple snapshots or wade through copious amounts of edge change logs, which are time-consuming especially when the query time interval is large. Since edge queries form the bedrock for graph queries, accelerating the speed of edge traversal is necessary to improve other temporal graph queries (e.g., subgraph query). Hence, exploring novel data structures that can improve traversal efficiency from time dimension becomes imperative to break through this bottleneck.

In this paper, we propose GeminiSketch, a novel sketch designed to address the aforementioned challenges in summarizing temporal graph streams. First, compared to existing graph summarization methods that compress graph streams into a single matrix, GeminiSketch partitions the matrix into two and uses them to store temporal edges for a given time interval respectively. For answering temporal graph queries, GeminiSketch can quickly locate the corresponding matrix based on the query time and directly access the storage information, which supports fast temporal graph queries. Secondly, to preserve the vertices that have hash collisions while ensuring memory efficient storage, GeminiSketch employs chain hashing to find vacant locations on the matrix to store temporal edges. The number of chain hashing calculations can be used as an indicator to measure the degree of matrix congestion, which facilitates matrix switchover to reduce hash collisions. Finally, GeminiSketch constructs a virtual bucket queue based on the access time of the buckets. The earlier the bucket is accessed, the farther ahead it is in the queue. In this way, the arrival orders of all edges are implicitly recorded in the virtual bucket queue. As time goes on, expired edges can be accurately and easily identified by *Rolling-out* strategy, which is a new elimination method proposed in this paper. This strategy enables the rapid elimination of expired edges, contributing to the overall effectiveness of GeminiSketch in managing temporal graph streams.

To the best of our knowledge, this is the first work to design a new summarization method for temporal graph streams. The main contributions of this paper can be summarized as follows:

• We propose GeminiSketch, a new data structure, to overcome the problems of accuracy and efficiency in summarizing temporal graph streams. It can support fast temporal graph queries, memory-efficient storage, and accurately expired edge elimination.

• We theoretically analyze the performance of GeminiSketch with regard to time and space complexity, and query accuracy.

• We conduct a series of experimental tests to demonstrate the performance of GeminiSketch with regard to the accuracy and efficiency of various temporal queries. The results show that GeminiSketch outperforms the state-of-the-art methods.

The remainder of this paper is organized as follows. Section 2 gives a review of related work. Section 3 provides problem definition. Section 4 describes the details of GeminiSketch. The mathematical analysis is provided in Section 5, followed by performance evaluations in Section 6. Finally, conclusions are drawn in the last section.

II. Related work

In this section, we review related work and discuss their limitations.

A. Temporal Graph Representation

The temporal characteristic of temporal graphs poses challenges for traditional graph representation methods because it requires to modify the graph structure driven by time evolving. There are two customized representation methods for temporal graphs: i) log-based: EdgeLog [12] can be regarded as a fundamental structure to represent temporal graphs. It stores additional time information for each edge in the adjacency list. Similarly, EveLog [13] stores change events of edges based on the adjacency list. CAS [13], TGCSA [14], and ck^d-tree [15] are compression versions of Evelog. DynamoGraph [16] is a distributed management system for temporal graphs that stores explicit connection information of vertices; ii) snapshot-based: Smo-index [17] is an indexing structure designed to facilitate answering temporal queries based on storing many snapshots and a log of translations between consecutive snapshots. To avoid full log scanning when answering temporal queries, HyperBit [18] builds some snapshots together with edge change logs to speed up the traversal. Clock-G [19] is a temporal graph management system that only records the difference between two successive snapshots. Although extensive research has been conducted on temporal graphs [1]–[3], [5], [37], [38], most existing methods do not address the problems of high memory consumption and slow query speed when encountering temporal graph streams. However, GeminiSketch offers new solutions to these problems.

TABLE I: Comparisons of GeminiSketch with other sketches

Method	Structure	ITI	Sliding Update	Insertion Cost	Space Cost	EC
TCM [25]	multiple matrices	○	○	$O(1)$	$O(\|E_s\|)$	$O(\|E_s\|)$
SBG [26]	multiple matrices	○	○	$O(1)$	$O(\|E_s\|)$	$O(\|E_s\|)$
GSS [30]	one matrix and an adjacency list	○	○	$O(1)$	$O(\|E_s\|)$	$O(\|E_s\|)$
Auxo [31]	tree-structured multiple matrices	○	○	$O(\log \|E_s\|)$	$O(\|E_s\| \log \|E_s\|)$	$O(\|E_s\|)$
ITeM [32]	one matrix with index tree	◐	●	$O(1)$	$O(W_s\|E_s\|)$	$O(\|E_s\|)$
PGSS [39]	one matrix with hierarchical array	●	○	$O(\log \|T_r\|)$	$O(\|E_s\| \log \|T_r\|)$	$O(\|E_s\|)$
Horae [33]	tree-structured multiple matrices	●	○	$O(\log \|T_r\|)$	$O(\|E_s\| \log \|T_r\|)$	$O(\|E_s\| \log \|T_r\|)$
HIGGS [36]	tree-structured multiple matrices	●	◐	$O(\log n_l)$	$O(\|E_s\| \log n_l)$	$O(n_l \|E_s\|)$
GeminiSketch	two matrices	●	●	$O(1)$	$O(\|E_s\|)$	$O(1)$

●: supported; ◐: partially supported; ○: not supported; ITI: Incorporating temporal information; EC: Expired edge elimination cost.
E_s: the number of edges in graph sketch; T_r: the length of the temporal range; n_l: the number of lead nodes; W_s: the number of windows.

B. Graph Stream Summarization

In order to reduce the scale of graph streams, the main idea of graph stream summarization is to use hash functions to compress vertices and aggregate the corresponding edges into one matrix, which can provide approximate graph queries. There are many efforts in designing new data structures for graph streams. They mainly differ in how to find an appropriate bucket for edges that have hash collisions and in how to accelerate the speed of graph queries. TCM [25] builds multiple matrices to independently store compressed vertices and edges. The query results are usually the minimum values among these matrices. Based on TCM, SBG [26] automatically balances the load of the multiple matrices according to the relative frequency of the edge labels. However, the main drawback of these two methods is high hash collisions. GSS [30] uses square hashing and fingerprints to avoid hash conflicts. It combines a compressed matrix and an adjacency list to store graph edges. Auxo [31] is a scalable structure by using a prefix embedded tree to accelerate update and query speed. Each node in the tree, which is a compressed matrix used to record edges, is dynamically generated based on the insertion status. However, they face an efficiency problem when dealing with temporal information.

Some methods have incorporated temporal information into the structure design. ITeM [32] equips each bucket in the compressed matrix with a fingerprint suffix index tree to improve graph queries when a hash collision occurs. It is a snapshot-based method that stores graph edges for each window independently and constructs a chain structure based on the start time of the window. As the window moves, the oldest ITeM will be deleted. PGSS [39] builds a hierarchical array to record edges with different time granularity in each bucket of GSS. Horae [33] is a multilayer structure constructed with different time prefix sizes. Each layer runs a GSS to record the graph edges under the specific time granularity. Both PGSS and Horae are inefficient in removing expired edges due to the hierarchical structure. HIGGS [36] employs a bottom-up approach to dynamically construct matrices that are arranged into a tree structure. The matrices in leaf nodes are used to directly store graph edges in each time interval, whereas the matrices in non-leaf nodes store aggregated edges and timestamps derived from their children. However, HIGGS needs to traverse many matrices to query a specific edge with different timestamps, degrading the query efficiency.

In summary, we compare GeminiSketch with other sketches in Table I and observe that GeminiSketch excels in all requirements. Most of them only focus on graph streams without temporal information even though they can achieve memory efficiency. The methods that incorporate temporal information have high cost in updating, memory usage, and expired edge elimination. Summarizing temporal graph streams is a challenging work since it needs to efficiently eliminate expired edges from a large number of edges as time goes on, while accurately retrieving graph queries. Therefore, we propose GeminiSketch to fulfill this gap by achieving fast graph queries, memory efficient storage, and rapid expired edge elimination.

III. Problem Definitions

In this section, we provide some definitions in temporal graph stream summarization.

Definition 1 (Temporal Graph Stream). A temporal graph stream is a continuous and time-evolving sequence of data items $S = \langle e_1, e_2, ..., e_x, ... \rangle$. Each item is represented as $e_x = (\langle s_x, d_x \rangle, w_x, t_x)$, indicating a directed edge from vertex s_x to vertex d_x at timestamp t_x with weight w_x. The edge e_x may appear multiple times at different timestamps. Therefore, a temporal graph can be modeled as $G = (V, E, T)$, where V is the set of vertices, E is the set of edges that appear within the given time T (known as active edges), and T is the temporal information that represents the lifetime of the graph. When T is a discrete timestamp, G is a graph snapshot at the time point T. When T is a time interval, G records the aggregation of multiple snapshots in the time interval T. The aggregation can be operations that sum, maximize, and minimize the weights of edges with the same vertices. We denote $|V|$ is the number of vertices and $|E|$ is the number of edges.

Definition 2 (Temporal Graph Sketch). Given a temporal graph $G = (V, E, T)$ formed by a temporal graph stream S within T, a temporal graph sketch $G_s = (V_s, E_s, T)$ is a compact summarization of G constructed by a hash function $F(\cdot)$ to downscale graphs, where V_s is the set of vertex hash results $\{F(v)|v \in V\}$ that satisfies $|V_s| \ll |V|$, E_s is the set of edge hash results $\{\langle F(s), F(d) \rangle | \langle s, d \rangle \in E\}$ that satisfies

$|E_s| \ll |E|$. Due to hash collisions, the weight of each edge in E_s is the cumulative weight of all conflicting edges in E.

Definition 3 (Temporal Graph Queries). Given a temporal graph $G = (V, E, T)$, there are three types of query primitives of temporal graphs: edge-related queries, vertex-related queries, and time-related queries [3]. All these queries need to specify a time interval $[t_b, t_e]$ within T. When $t_b = t_e$, queries are classified as point-time queries. When $t_b < t_e$, we refer to them as interval-time queries.

• Edge-related queries: query edge information during $[t_b, t_e]$, e.g., checking whether there is an edge within $[t_b, t_e]$.

• Vertex-related queries: query vertex information during $[t_b, t_e]$, e.g., finding neighbors of vertex v or whether $v \in V$ within $[t_b, t_e]$.

• Time-related queries: query general information of temporal graph during $[t_b, t_e]$, e.g., finding all active edges within $[t_b, t_e]$, checking relationship among vertices in $[t_b, t_e]$.

Based on these primitives, we can obtain all the information from the temporal graph, e.g., active subgraphs, reachable path, shortest paths among vertices, persistent community.

IV. GeminiSketch Design

In this section, we discuss the data structure and basic operations of GeminiSketch.

A. GeminiSketch Overview

As shown in Figure 1, GeminiSketch contains two $n \times n$ matrices, denoted as G_1 and G_2. They are alternately used to store temporal graphs, named working matrix and idle matrix. When the working matrix nears capacity, we will freeze it and shift the edge update to the idle one. The dual-matrix design, whose memory usage is equivalent to a single matrix of current methods, stems from observations of temporal graph streams where recent edges dominate while older ones expire. The single matrix accumulates expired edges, increasing hash collisions and inefficient expired edge elimination.

Each matrix has a working status (WS), a global timestamp (GT), and three global pointers (HP, TP, and MP). The WS represents the working status of the matrix, where "0" denotes the matrix is working and "1" denotes the matrix is idle. The GT records the most recent timestamp of the edges stored in the matrix, which is used to accelerate the speed of graph queries and help to eliminate expired edges. The HP and TP act as head and tail pointers of a virtual bucket queue, which is constructed based on the accessed time of the bucket in each matrix. The virtual bucket queue is used to implicitly reserve the arrival orders for all edges. The oldest edges are eliminated from the bucket pointed by HP, whereas completely new edges are inserted into the bucket indicated by TP. MP points to the bucket that marks the end position of each round of expired edge elimination. This design enables GeminiSketch to quickly eliminate expired edges rather than traverse all buckets, which supports real-time management of the temporal graph. In particular, we construct the virtual bucket queue to take advantage of the queue to quickly determine the first accessed bucket. However, we can delete any bucket in the virtual queue based on the status of the bucket.

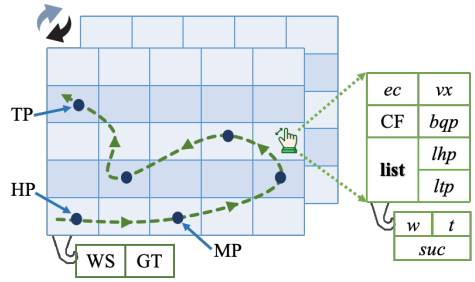


Fig. 1: The structure of GeminiSketch.

The bucket in the matrix is represented as $G_{\{1,2\}}[i][j](1 \leq i, j \leq n)$. We use $G[i][j]$ for simple representation. Each bucket maintains: (i) three variable fields $\{ec, vx, \text{CF}\}$, where ec is a counter that records the number of edges stored in the bucket, vx are the source and destination vertices of the edge, CF is a flag that denotes whether the bucket is the last one in the hash chain; (ii) a list that stores edges. Each element in the list represents an edge and is expressed as $\{w, t, suc\}$, where w is the edge weight, t is the timestamp, and suc points to successor element in the list; (iii) three pointers $\{bqp, lhp, ltp\}$, where bqp is a pointer of virtual bucket queue that points to the successor bucket, lhp and ltp are the head pointer and tail pointer that point to the first and last element of the list.

GeminiSketch utilizes two kinds of hash functions. The first is the hash function $F(\cdot)$, whose value range is $[0, N)$. It transforms a temporal graph stream into the corresponding temporal graph sketch. The second is chain hashing $H(\cdot)$, which is applied to pinpoint an appropriate bucket within the matrix. It contains a set of hash functions $\{h_1(\cdot), h_2(\cdot), ..., h_r(\cdot)\}$ with value range $[0, n)$, r is not a large value but is sufficient to locate the bucket. For edge $e = \langle s, d \rangle$, we first calculate $\tilde{s} = F(s)$ and $\tilde{d} = F(d)$ as its location in the temporal graph sketch. Then, we use h_1 to find the bucket $G[h_1(\tilde{s})][h_1(\tilde{d})]$. If it has been occupied by another edge, we use h_2 to find another bucket. The hashing process continues until the appropriate bucket (empty or occupied by $\langle \tilde{s}, \tilde{d} \rangle$) is found. The hash results of $\tilde{s}$ (similarly for $\tilde{d}$) are calculated as follows:

$$\begin{cases} h_1(\tilde{s}) = (q \times \tilde{s} + p)\%n, c = 1 \\ h_c(\tilde{s}) = (q \times h_{c-1}(\tilde{s}) + p)\%n, 2 \leq c \leq r \end{cases}$$

where q and p are selected primer numbers. The hash functions in $H(\cdot)$ are independent with each other, as proven in [40].

B. Update Operations

Initially, GeminiSketch is empty and has the following three update operations to summarize temporal graph streams.

Insertion. As described in Algorithm 1, for edge $e = (\langle s, d \rangle, w, t)$, we first find the working matrix by checking whose WS is 0. Then we locate the appropriate bucket $G[H(\tilde{s})][H(\tilde{d})]$. During the locating process guided by chain hashing, we set the CF fields of the buckets that are occupied by other edges to 0. There are two cases:

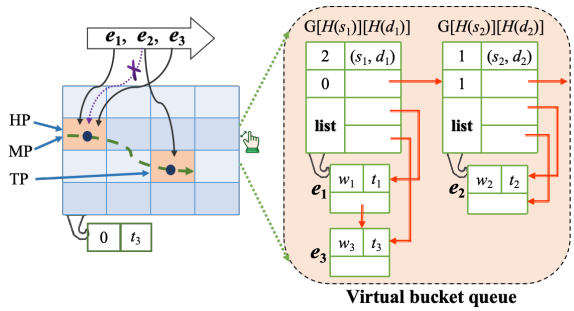


Fig. 2: Insertion operation of GeminiSketch.

1) $G[H(\tilde{s})][H(\tilde{d})]$ is empty, we increase ec by 1, update vx, set CF to 1, add e to $G[H(\tilde{s})][H(\tilde{d})].list$ as the first element, update GT of the working matrix to t. If $G[H(\tilde{s})][H(\tilde{d})]$ is the first updated bucket of the matrix, HP, MP and TP should point to it; otherwise, insert it to the tail of virtual bucket queue by visiting TP;

2) $G[H(\tilde{s})][H(\tilde{d})]$ has been occupied, the new edge should be added to the end of $G[H(\tilde{s})][H(\tilde{d})].list$, update ec and GT. In this case, $G[H(\tilde{s})][H(\tilde{d})]$ stores the latest edge and should be shifted to the tail of the virtual bucket queue. However, the edges from the vertex $\tilde{s}$ to $\tilde{d}$ may arrive at different times. If we simply shift the bucket to the tail, the timeline of the virtual bucket queue is out of order since the edges that arrived early are moved to the back. Therefore, we leave $G[H(\tilde{s})][H(\tilde{d})]$ in the original position to maintain its access order.

Figure 2 gives an example of edge insertion. Given three edges arrive sequentially, denoted $e_1 = (\langle s_1, d_1\rangle, w_1, t_1)$, $e_2 = (\langle s_2, d_2\rangle, w_2, t_2)$, and $e_3 = (\langle s_1, d_1\rangle, w_3, t_3)$. e_1 is hashed into $G[H(\tilde{s_1})][H(\tilde{d_1})]$ which is the first updated bucket of the matrix. Thus, we update the fields of the bucket and make HP, MP, and TP point to it. e_2 is mapped to $G[H(\tilde{s_1})][H(\tilde{d_1})]$ but it is occupied by e_1. Therefore, we first set $G[H(\tilde{s_1})][H(\tilde{d_1})]$.CF to 0 and then find another empty bucket $G[H(\tilde{s_2})][H(\tilde{d_2})]$. TP moves to $G[H(\tilde{s_2})][H(\tilde{d_2})]$ and GT updates to t_2. e_3 has same vertices with e_1 and we add e_3 to $G[H(\tilde{s_1})][H(\tilde{d_1})]$, increase ec, and update GT to t_3.

Expired edge elimination. When the edges that arrived early fall outside the time window, expired edges need to be removed one by one from the head of the virtual bucket queue. Since multiple edges with the same vertices but different timestamps are stored in a single bucket, only the first-arrived edge adheres to the principle of bucket queue. For example, the second element in a particular bucket may arrive much later than the first element in its successor bucket, meaning that it is still active. Consequently, to prevent accidental removal of non-expired edges, we propose a new expired edge elimination method, called *Rolling-out* strategy. It allows us to pinpoint true expired edges in a wavering manner along the virtual bucket queue, rather than simply directly remove all the edges in the foremost bucket at once.

Rolling-out strategy: At the beginning, we scan all edges of the bucket positioned at the head of the virtual bucket queue. For each edge encountered, we perform a check: If the edge is expired, we promptly remove it from the list and decrease ec by 1. Conversely, if the edge is active, we bypass this particular bucket and immediately check its successor bucket. As we progress through the buckets, when ec of a bucket reaches 0, we dequeue that bucket from the virtual bucket queue and initialize it, preparing it for future use. This elimination process persists until we encounter a bucket whose first edge is non-expired. After that, we make MP point to that bucket. As time goes on, the next round of eliminating expired edges is triggered when the first edge of MP is out of the window. In response, MP will move to the next non-expired bucket accordingly. We continuously repeat the above process to remove expired edges, as described in Algorithm 2.

Algorithm 1 : Insertion operation of GeminiSketch

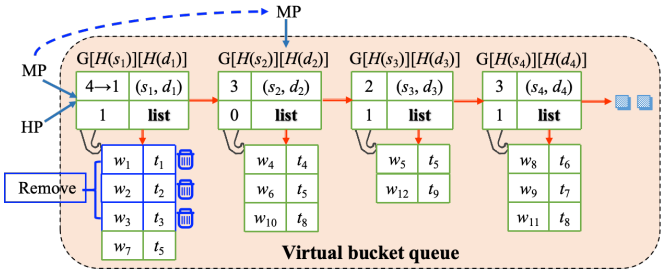
Input: $e = (\langle s, d\rangle, w, t)$, GeminiSketch.
Output: Updated GeminiSketch.

```
Find the working matrix G;
s̃, d̃ = F(s), F(d);
for (c = 1; c ≤ r; c++):
    i, j = h_c(s̃), h_c(d̃);
    if G[i][j].vx = ⟨s, d⟩ || G[i][j] = NULL:
        break;
    else:
        G[i][j].CF = 0;
if G[i][j] = NULL:
    G[i][j].ec += 1;
    G[i][j].vx = ⟨s, d⟩;
    G[i][j].CF = 1;
    Insert e to G[i][j].list;
    G.GT = t;
    if G[i][j] is the first updated bucket of G:
        HP, MP, TP point to G[i][j];
    else:
        Insert G[i][j] to the tail of virtual bucket queue;
else:
    G[i][j].ec += 1;
    Insert e to G[i][j].list;
    G.GT = t
```

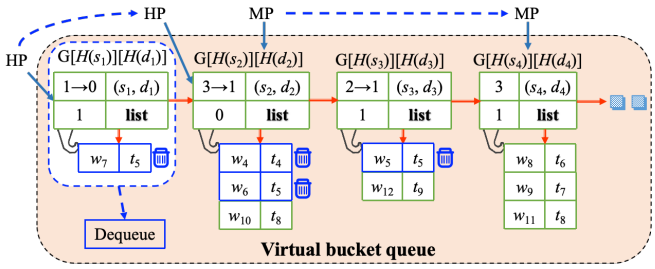
Figure 3 provides an example of the expired edge elimination process employing the *Rolling-out* strategy. For the sake of clarity, the representation of buckets has been simplified.

• First round (Figure 3 (a)): Given that t_4 is the elimination time, we traverse the bucket from the head of the virtual bucket queue. During this process, we first identify three expired edges in $G[H(s_1)][H(d_1)]$ and then remove them from the edge list. Next, we find that the first element of the successor bucket is non-expired. Thus, we direct MP to point towards $G[H(s_2)][H(d_2)]$.

• Second round (Figure 3 (b)): Suppose that the elimination time is t_6, the first edge in the bucket to which MP points is outside the time window, triggering the second round of elimination. Since the last edge in $G[H(s_1)][H(d_1)]$ has been removed, we dequeue $G[H(s_1)][H(d_1)]$ from the queue and reposition HP to point at its successor bucket. Similarly, we



(a) First round: elimination time is t_4.



(b) Second round: elimination time is t_6.

Fig. 3: Example of running *Rolling-out* strategy.

identify and eliminate two edges from $G[H(s_2)][H(d_2)]$ and one edge from $G[H(s_3)][H(d_3)]$. Finally, we adjust MP to point to $G[H(s_4)][H(d_4)]$.

Matrix switchover. This operation serves to switch between the working matrix and the idle matrix. The congestion status of the working matrix has a significant influence on the number of chain hashing computations. As the matrix becomes increasingly crowded, the number of hash operations necessary to find the appropriate bucket also increases. In such a situation, the time overhead associated with updating surpasses the space overhead. To address this issue, we should freeze the working matrix and activate the idle one with the aim of reducing the time consumption during updates. Many counting methods can be used to achieve our goal. To accurately assess the level of congestion, we maintain a short queue of length l, which records the number of hash calculations for the most recent l updates. If the average number of hash calculations exceeds a predefined threshold, we will carry out the matrix switchover operation and update the WS of both matrices. Simultaneously, the *Rolling-out* strategy continues to operate on the frozen matrix until all expired edges within it have been eliminated. Once this is accomplished, the elimination process then shifts to the working matrix.

C. Query Operations

For all types of temporal queries, we should first find the matrix that stores their corresponding graph information according to the query time. Given a query with a time interval $[t_b, t_e]$ and assuming that the minimum GT among the two matrices is T_m, there are three cases: 1) if $t_e \leq T_m$, we perform the query operation on the matrix that has T_m; 2) if $T_m < t_b$, we perform graph query on other matrix; 3) if $t_b \leq T_m \leq t_e$,

Algorithm 2 : Expired edge elimination of GeminiSketch

Input: Elimination time T_e, GeminiSketch.
Output: GeminiSketch with no expired edges.

```
if MP.list.lhp.t ≤ T_e:
    WP = HP;
    while WP ≠ MP:
        while (WP.list.lhp.t ≤ T_e):
            WP.ec -= 1 ;
            if WP.ec = 0:
                NB = WP;
                WP = WP.bqp;
                Remove NB from the virtual bucket queue;
            else:
                WP.list.lhp = WP.list.lhp.suc;
        WP = WP.bqp:
    while (MP.list.lhp.t ≤ T_e):
        S = MP;
        while (S.list.lhp.t ≤ T_e):
            S.ec -= 1;
            if S.ec = 0:
                NB = S;
                MP = MP.bqp;
                Remove NB from the virtual bucket queue;
            else:
                S.list.lhp = S.list.lhp.suc;
        MP = MP.bqp;
```

Algorithm 3 : Temporal edge query of GeminiSketch

Input: $Q_e = (\langle s, d\rangle, [t_b, t_e])$, GeminiSketch.
Output: The queried edge information.

```
Calculate H(s̃), H(d̃);
if G[H(s̃)][H(d̃)] is empty:
    return False;
else:
    weight = 0
    for each edge e in G[H(s̃)][H(d̃)].list:
        if t_b ≤ e.t ≤ t_e:
            weight += e.w
    if weight = 0:
        return False;
    else:
        return weight
```

we first split the query time interval into $[t_b, T_m]$ and $(T_m, t_e]$, and then conduct the query on both two matrices. The final query is the integration of two returned queries. After finding the corresponding matrix, we can perform the following basic graph queries on GeminiSketch.

Temporal edge query. As described in Algorithm 3, for query $Q_e = (\langle s, d\rangle, [t_b, t_e])$, we map the edge $\langle s, d\rangle$ to $G[H(\tilde{s})][H(\tilde{d})]$ and traverse $G[H(\tilde{s})][H(\tilde{d})].list$ to judge whether the edge is active within $[t_b, t_e]$. If it does, we return the aggregated edge weight. Otherwise, we report False. Additionally, GeminiSketch can rapidly estimate the statistics

Algorithm 4 : Temporal vertex query of GeminiSketch

Input: $Q_e = (v, [t_b, t_e])$, GeminiSketch.
Output: The queried vertex information.

```
1: ṽ = F(v);
2: for (c = 1; c ≤ r; c++):
3:     i = h_c(ṽ);
4:     for (j = 1; j ≤ n; j++):
5:         if G[i][j] contains v:
6:             return True;
7:         else:
8:             return False;
```

of active edges within a time interval, which is not well supported by other work. Given a time interval, we can find the edges by traversing the buckets one by one from the head of virtual bucket queue and perform estimation, e.g., the number of active edges or the number of different active edges.

Temporal vertex query. As described in Algorithm 4, for query $Q_v = (v, [t_b, t_e])$, we obtain r rows (columns) of v by computing $H(\tilde{v})$ and travel through all the buckets in these rows (columns) to find the buckets that contain v. If there is no bucket having v, we return False. Otherwise, we report True. We can also access their edge list and select the active edges by comparing the timestamp with $[t_b, t_e]$. We add picked edges to the query set and support many derived queries, e.g., the total outgoing (incoming) weight of vertex v and the number of outgoing (incoming) edges of v within $[t_b, t_e]$.

Temporal subgraph query. For subgraph query $Q_s = (\langle s_1, d_1\rangle, .., \langle s_k, d_k\rangle, [t_b, t_e])$, we run Algorithm 1 for each edge in the subgraph. If there is an edge query that returns False, we report False to represent that the subgraph is not exactly matched in the queried time interval. Otherwise, we sum up the queried edge weights as the query result. We can also give the heavy edges in the subgraph.

Temporal reachability query. For reachability query $Q_r = (\langle s, d\rangle, [t_b, t_e])$, we can adopt any graph traversal method by imposing a time constraint on the path search to find whether there is a path from s to d. If it has, we can return True. Otherwise, we report False to denote d is unreachable from s in the time interval $[t_b, t_e]$.

Compensation of chain cut-off. Due to hash collisions, the hash results of different vertices may be identical, causing their hash chains to overlap. Suppose $G[H(\tilde{s_1})][H(\tilde{d_1})]$ is a bucket in the hash chain of $\langle s_2, d_2\rangle$ and lies outside the time window. When running *Rolling-out* strategy, $G[H(\tilde{s_1})][H(\tilde{d_1})]$ is initialized, causing a hash chain cut-off of $\langle s_2, d_2\rangle$. This situation leads to false negative when querying the information of $\langle s_2, d_2\rangle$. In order to compensate for this problem, when encountering an empty bucket during matching buckets on the hash chain, we continue to check g positions by using chain hashing, instead of simply returning False.

V. Theoretical Analysis

In this section, we analyze the time and space complexity, and query accuracy of GeminiSketch.

A. Time and Space Complexity

Time complexity: Because inserting each edge requires finding at most r buckets and updating several fields with $O(1)$ complexity, the update time complexity is $O(1)$ since r is not a very large value. In the process of eliminating expired edges, we begin at the head of the virtual bucket queue rather than scanning all $n \times n$ buckets. Therefore, the elimination of expired edges achieves $O(1)$ time complexity. The time complexity of the edge query has the same time complexity as that of the insertion operation. A vertex query needs to traverse r rows (columns) with n buckets, which has $O(rn)$ complexity. A subgraph query with k edges involves performing a query for each edge, so the time complexity is $O(rk)$. A reachability query often necessitates multiple vertex queries. In the best-case, the time complexity is $O(rn)$. In the worst-case, where we need to perform vertex queries for a large subset of V_s, the time complexity escalates to $O(rn|V_s|)$, where $|V_s|$ is the number of vertices in G_s. In summary, GeminiSketch demonstrates efficient time complexity characteristics for various operations, making it suitable for handling temporal graph streams with reasonable computational resources.

Space complexity: GeminiSketch records the edges of a temporal graph sketch by additionally utilizing some fields to realize insertion, expired edge elimination, matrix switchover, and graph query retrieval. The memory usage of these fields is much lower than the memory usage of storing edges of the temporal graph sketch. Therefore, the space complexity of the GeminiSketch is $O(|E_s|)$, where $|E_s|$ is the number of edges in the temporal graph sketch G_s and is determined by the value range of the hash function $F(\cdot)$.

B. Query Accuracy Analysis

GeminiSketch performs two steps to summarize temporal graph streams. In the first step, it uses $F(\cdot)$ to transform a temporal graph stream to a temporal graph sketch. The errors of this step brought to temporal graph queries are caused by hash collisions, which have been discussed in detail in [30]. Therefore, we omit the error proof of the first step. In the second step, GeminiSketch uses chain hashing $H(\cdot)$ to find appropriate buckets to store graph information. This step contains errors coming from the chain cut-off problem, which causes temporal query failure. Thus, in the following, we mainly analyze the query failure error in the second step.

Given a temporal graph sketch $G_s = (V_s, E_s, T)$, $\tilde{e}$ is a temporal edge, $E_1(\tilde{e})$ is the set of edges that have the same source or destination vertex as $\tilde{e}$, $E_2(\tilde{e})$ is the set of edges that do not share the same vertices as $\tilde{e}$. For each edge in $E_1(\tilde{e})$, the probability that they have no collision is $\mathrm{P}_1 = 1 - \frac{1}{n}$. For each edge in $E_2(\tilde{e})$, the probability that it is not stored in the same bucket is $\mathrm{P}_2 = (1 - \frac{1}{n})^2$. Suppose $\tilde{e}$ is stored in (y+1)-th bucket on the hash chain. If $\tilde{e}$ collides with an edge in one of the y buckets that lies ahead and the edge is removed before $\tilde{e}$, the chain cut-off occurs. The probability that $\tilde{e}$ collides with the edges of $E_1(\tilde{e})$ at least once is

$$\bar{\mathrm{P}}_1 = 1 - ((1 - \frac{1}{n})^y)^{|E_1(\tilde{e})|} = 1 - e^{-\frac{y|E_1(\tilde{e})|}{n}}$$

Similarly, the probability that $\tilde{e}$ collides with the edges of $E_2(\tilde{e})$ at least once is

$$\bar{\mathrm{P}}_2 = 1 - ((1 - \frac{1}{n})^{2y})^{|E_2(\tilde{e})|} = 1 - e^{-\frac{2y|E_2(\tilde{e})|}{n}}$$

Therefore, the probability that chain cut-off occurs is $\mathrm{P}_{cut} = \bar{\mathrm{P}}_1 \times \bar{\mathrm{P}}_2$. As $y < r \ll n$, P_{cut} is mainly influenced by $|E_1(\tilde{e})|$ and $|E_2(\tilde{e})|$, but is still a small value. Furthermore, in the skewed distribution scenario, if $\tilde{e}$ is a heavy edge, it will be mainly affected by the edges in $E_2(\tilde{e})$. But we use *Rolling-out* elimination strategy to timely remove expired edges (mostly light edges), the above affect becoming small. Actually, the temporal edges arrive out of order, and thus the process of removing expired edges is also irregular, which reduces P_{cut}. To address the chain cut-off problem, we have proposed the compensation method in Subsection 4.3, which has been proven effective in performance evaluation.

VI. Performance Evaluation

In this section, we perform a series of experiments to show the performance of GeminiSketch with respect to ablation study (§VI.B), query accuracy (§VI.C), efficiency (§VI.D), and case study on Neo4j (§VI.E). We begin by describing experimental settings. Next, experimental results on performance evaluations are reported. We release the source code of GeminiSketch on GitHub [41].

A. Experimental Setting

Experiment platform: All experiments are conducted using C++ on a machine with a 10-core processor (20 threads, Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz) and 62 GB DRAM memory.

Datasets: To construct temporal graph streams, we use four real-world temporal graph datasets:

• Stackoverflow [42]: A temporal network of interactions on the stack exchange web site Stack Overflow, which contains $2,601,977$ vertices and $63,497,050$ temporal edges.

• Wiki [43]: A temporal network representing Wikipedia users editing each other's Talk page, which contains $1,140,149$ vertices and $7,833,140$ temporal edges.

• Reddit [44]: The hyperlink network represents the directed connections between two subreddits, which contains $55,863$ vertices and $858,490$ temporal edges.

• Super User [45]: A temporal network of interactions on the stack exchange website, which contains $194,085$ vertices and $1,443,339$ temporal edges.

Evaluation metrics: We use the following metrics:

• Average Relative Error (ARE): It is used to measure the accuracy of queried weights in the tasks of edge query, vertex query, and subgraph query. ARE is calculated as $\frac{1}{|q|}\sum_{i=1}^{|q|}|\hat{q}_i - q_i|/q_i$, where $\hat{q}_i$ is queried weight, q_i is true weight, and $|q|$ is the number of queries.

• Average Precision: It is used to measure the accuracy of reachability query. Since graph sketch has only false positives, the average precision is calculated as $\frac{1}{z}\sum_{i=1}^{z}|r_i|/|\hat{z}_i|$, where z_i is the accurate set of reachable vertices, $\hat{z}_i$ is the set of queried reachable vertices, z is the size of query set.

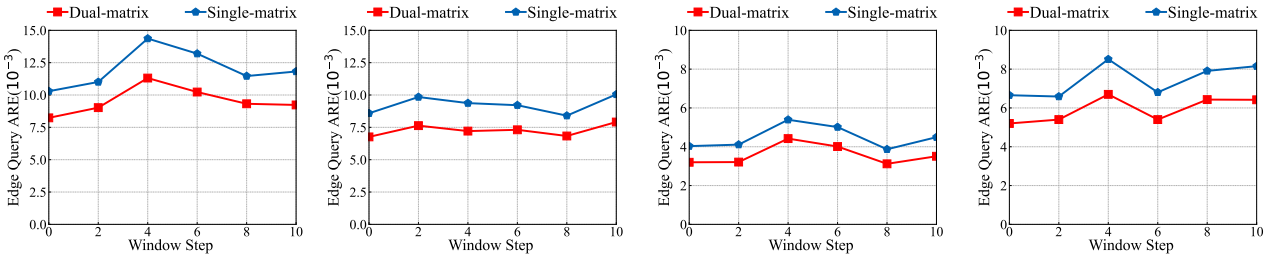
• Throughput: We use million operations per second (Mops) to evaluate the throughput of all methods.

Baselines: We compare GeminiSketch with current sketches that exhibit excellent performance in graph stream summarization, including TCM [25], GSS [30], SBG [26], Auxo [31], Horae [33], ITeM [32], and HIGGS [36]. Since TCM, GSS, SBG, and Auxo lack direct support for temporal range queries, we enhance them using the temporal range decomposition scheme [33], denoted as TCM+time, GSS+time, SBG+time, and Auxo+time. We also compare GeminiSketch with current indexing structures for temporal graph streams on efficiency, including EdgeLog [12], Evelog [13], Smo-index [17], and WBIndex with timestamps (WBIndex+time) [4]. For all methods, the memory budget is fixed at 20MB to ensure a uniform experimental setting, and the parameters are set according to the original papers. For GeminiSketch, expiration threshold (the maximum allowed time difference between current time and edge time) is set to 100 days based on the time spans of the datasets. The conflict threshold k in the matrix switchover operation and the length of hash chain r are set to 20. But only a very small portion of the edges will be hashed more than ten times. The length of the short queue is set to 10.

Temporal query construction: Due to the characteristics of temporal graph streams, it is difficult to process entire data at once. Therefore, we divide the streams into fixed-size and non-overlapping windows to efficiently analyze data while controlling memory usage. We set each window size to 50,000. However, existing methods lack an effective strategy to eliminate expired edges as the window moves. To ensure a fair comparison, we reinitialize their counting structures in each new window to ensure that the query results only contain graph information within that window. This re-initialization process is necessary since the methods may accumulate expired edges, degrading accuracy and efficiency over time. For every temporal query, we adjust the query range within a span of 100 days. We randomly generate $10,000$ edge queries and 5000 vertex queries. For subgraph query, we set the subgraph size from 50 to 200 and generate 1000 queries for each subgraph size. For path query, we set the path length from 1 to 10 and construct 1000 queries for each path length. All reported results represent the average of 1000 runs.

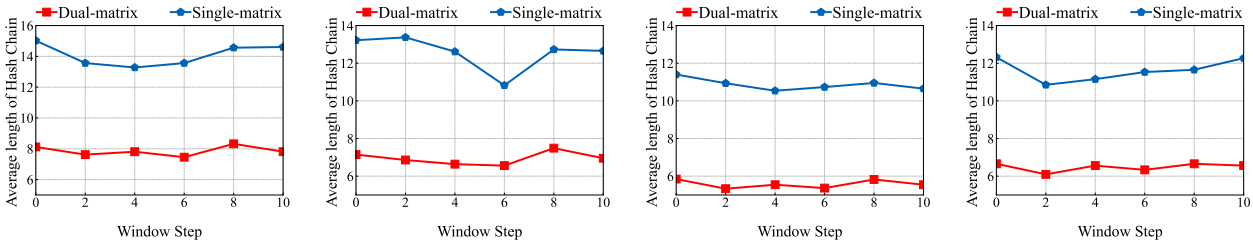
B. Evaluations on Ablation Study

The effectiveness of dual-matrix design: We compare our dual-matrix design with the single-matrix design in accuracy and efficiency of edge query under the same memory (the results are similar for other queries). The differences between them lie only in the size of the matrix and whether the matrix switchover operation is used. Figure 4 and Figure 5 show the ARE of the edge query and the average length of the hash chain of both designs. The x-axis "Window Step" indicates the incremental movement of the time window along the temporal dimension. The results indicate that, compared to the single-matrix design, our dual-matrix design reduces the error by an average of 26% and shortens the hash chain length by approximately 70% to 108%.



(a) **Stackoverflow** (b) **Wiki** (c) **Reddit** (d) **Super User**

Fig. 4: Accuracy of dual-matrix and single-matrix design.



(a) **Stackoverflow** (b) **Wiki** (c) **Reddit** (d) **Super User**

Fig. 5: Efficiency of dual-matrix and single-matrix design.

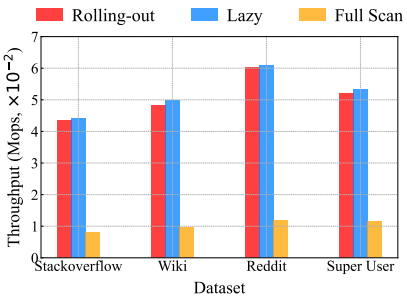


Fig. 6: Throughput of GeminiSketch with three strategies.

The effectiveness of elimination strategy: We compare accuracy and efficiency of three elimination strategies on edge query, namely the *Rolling-out* elimination strategy proposed in this paper, the full-scan elimination strategy that scans the entire matrix per update, and the lazy elimination strategy that only scans the bucket accessed by an incoming edge. Figure 6 shows that our strategy and lazy strategy are around five times faster than the full-scan strategy. Although the speed of the lazy strategy is similar to our strategy, its delayed edge elimination introduces query errors, as shown in Table II.

The effectiveness of chain cut-off compensation: Because chain cut-off problem is the main error in GeminiSketch, we test the impact of g in the chain cut-off compensation on edge query accuracy, as shown in Figure 7. Our findings reveal that when $g = 1$, the ARE shows an improvement of

TABLE II: ARE of edge query of three elimination strategies.

Strategy	SO	Wiki	Reddit	SU
Rolling-out	**0.0168**	**0.00698**	**0.00356**	**0.00545**
Lazy	0.0323	0.01152	0.00649	0.00904
Full-scan	0.0171	0.00721	0.00361	0.00527

SO: Stackoverflow; SU: Super User.

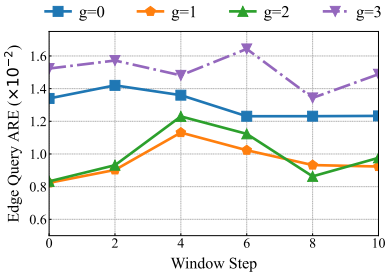


Fig. 7: Results of chain cut-off compensation.

approximately $16\% - 21\%$ compared to $g = 0$. However, no further improvement increases as g increases, since additional errors are introduced. Based on the trade-off between accuracy and computational cost, we set g = 1 as the default value.

C. Evaluations on Query Accuracy

Accuracy on edge query: Figure 8 illustrates the ARE of temporal edge queries of graph sketches. GeminiSketch achieves the smallest ARE compared to others. For example,

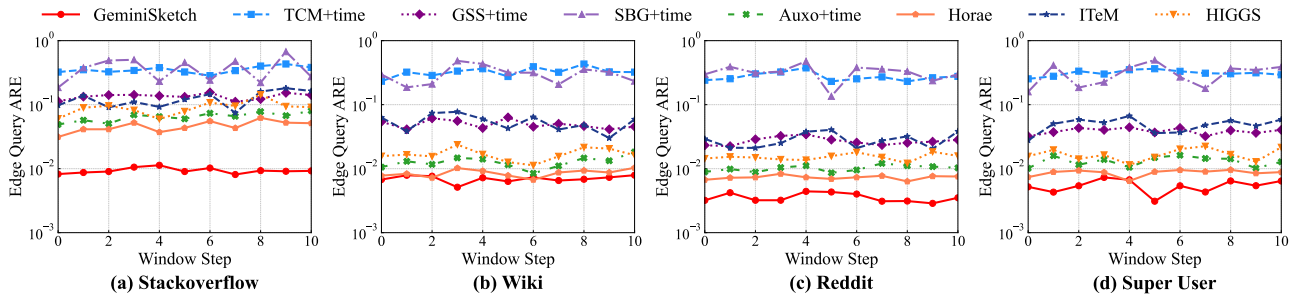


Fig. 8: Average relative error of edge query.

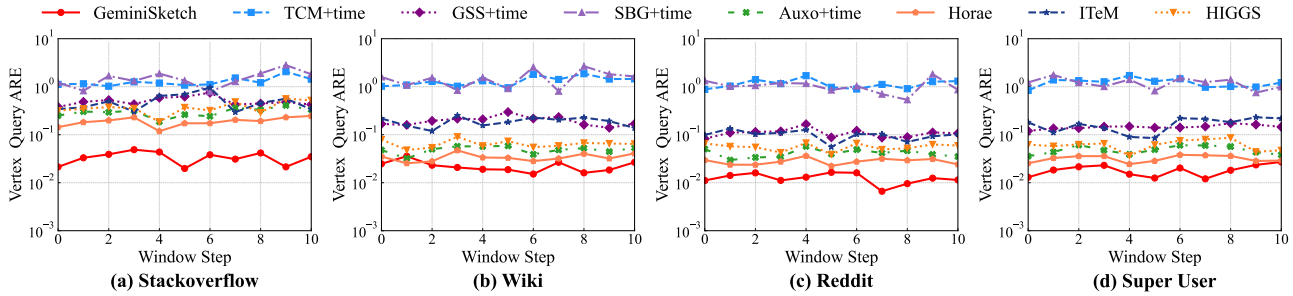


Fig. 9: Average relative error of vertex query.

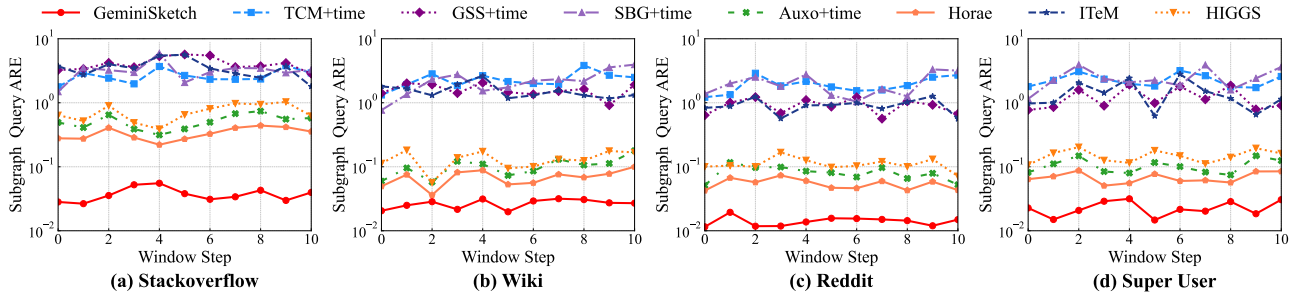


Fig. 10: Average relative error of subgraph query.

on Stackoverflow, GeminiSketch has 3.59, 6.52, 10.99, 11.51, 12.92, 36.08, 37.12 times lower ARE than Horae, Auxo, HIGGS, ITeM, GSS, TCM, SBG. Although Horae is comparable to GeminiSketch on Wiki, it requires re-initialization in each window, leading to additional time consumption.

Accuracy on vertex query: Figure 9 presents the ARE of temporal vertex queries in four datasets. GeminiSketch achieves an average of 11.93 times lower ARE than others. Horae has a similar performance to GeminiSketch on some datasets since they both consider temporal information. However, on Stackoverflow, GeminiSketch performs well since Stackoverflow has the longest time span among all datasets.

Accuracy on subgraph query: Figure 10 provides that the ARE of GeminiSketch demonstrates more significant advantages than other methods, where it maintains a small ARE below 0.1. The reason is that performing subgraph queries needs to consider the complex relationships and temporal information between multiple edges, which requires an accurate capture of the structure and topology of the subgraph. GeminiSketch constructs a virtual bucket queue to implicitly store temporal information in a complete and ordered manner, giving it a greater advantage in subgraph query.

Accuracy on reachability query: Figure 11 displays the average precision of path reachability queries. The presence of a single incorrect edge on a path indicates that the final result is wrong. For the same reasons as the subgraph query outlined previously, GeminiSketch can achieve relatively good results.

D. Evaluations on Efficiency

Query time: Figure 12 and Figure 13 demonstrates that GeminiSketch has a significant advantage in query efficiency

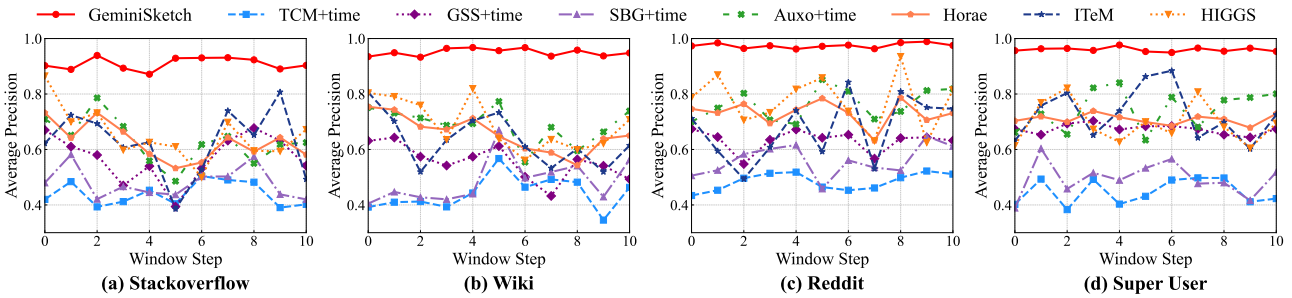


Fig. 11: Average precision of reachability query.

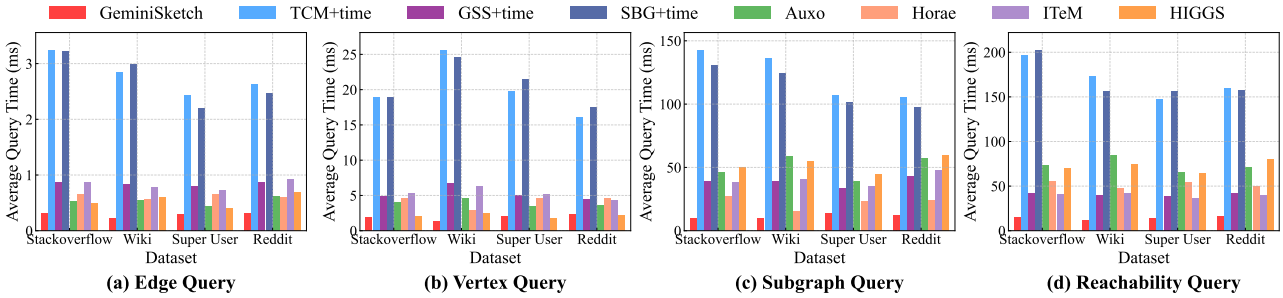


Fig. 12: Average query time of graph sketches.

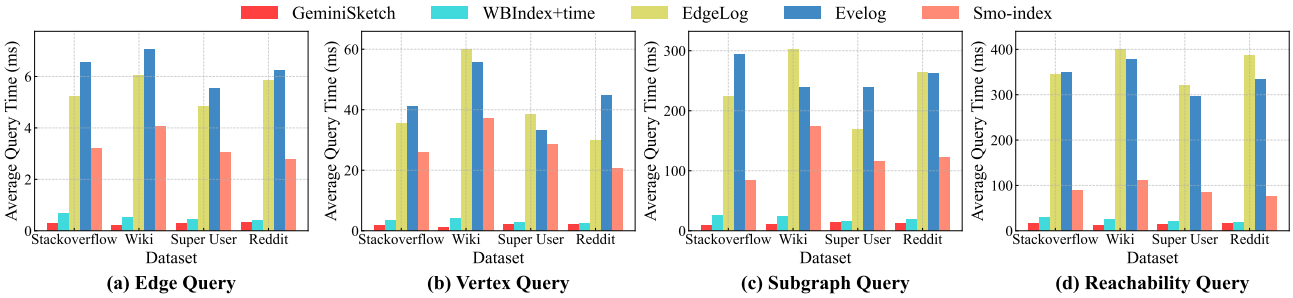


Fig. 13: Average query time of temporal graph representation methods.

compared to both graph sketches and temporal graph representation methods, especially in subgraph and reachability queries. Compared to other methods, GeminiSketch uses two independent matrices to summarize edges that have same vertexes but different timestamps for a given time interval respectively, achieving fast temporal graph queries.

Delay time: In our experiments, we should initialize each method in every new window since they do not support sliding analysis, which causes additional time consumption. Therefore, we define the initialization time as the delay time to show the insertion efficiency. As shown in Table III, GeminiSketch achieves zero delay time on all datasets since it efficiently eliminates expired edges without requiring structure re-initialization, demonstrating superior temporal efficiency and enabling real-time queries.

TABLE III: Delay time comparison as window moves (ms).

Method	SO	Wiki	Reddit	SU
GeminiSketch	**0.00**	**0.00**	**0.00**	**0.00**
TCM+time	1032.12	1054.27	1034.54	973.54
GSS+time	321.21	287.22	223.13	253.28
SGB+time	943.52	1001.4	976.21	989.45
Auxo+time	87.21	71.24	91.59	97.31
Horae	123.81	135.76	167.53	153.27
ITeM	267.31	211.43	192.73	231.42
HIGGS	79.03	63.53	53.21	78.95
WBIndex+time	432.21	446.17	463.44	397.64
EdgeLog	34.32	48.56	43.73	36.21
Evelog	45.21	47.24	42.56	39.16
Smo-index	26.37	23.45	19.23	28.24

SO: Stackoverflow; SU: Super User.

Throughput: As shown in Figure 14, GeminiSketch demonstrates significant advantages in the insertion process com-

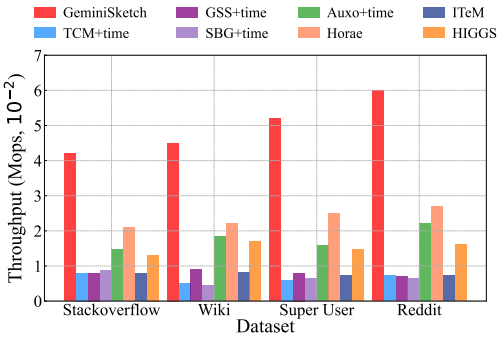


Fig. 14: Throughput of graph sketches.

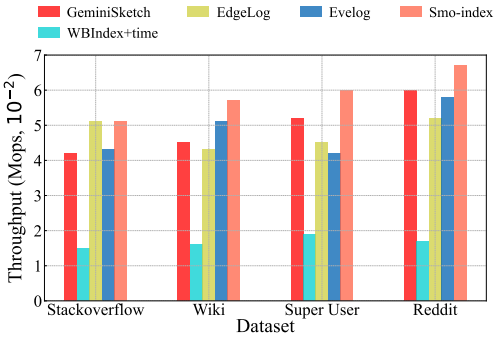
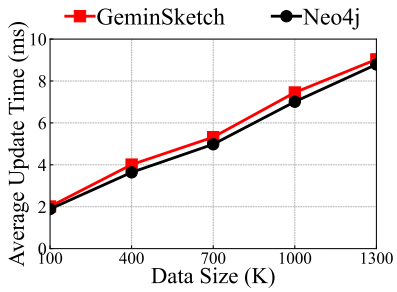
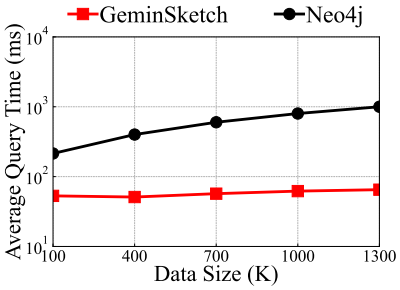


Fig. 15: Throughput of temporal graph representation methods.



(a) Update time



(b) Edge query time

Fig. 16: Results of efficiency of GeminiSketch and Neo4j.

pared to other graph sketches, while Figure 15 also shows a comparable time performance to temporal graph representation methods. The efficiency comes from the enhanced ability of GeminiSketch to effectively mitigate hash collisions through its switch mechanism.

E. Evaluations on Neo4j Database

We further compare GeminiSketch against Neo4j [46], a widely adopted real-world graph database. As illustrated in Figure 16, GeminiSketch consistently outperforms the original Neo4j in average query times, demonstrating at least a 4.32 times speedup. This performance gap widens with increasing data volumes. For example, for 1300K data size, GeminiSketch achieves an average query time of 6.52 milliseconds, which makes it 15.38 times faster than Neo4j. The insertion of GeminiSketch is averaged 5.564 milliseconds, which is 10.03% longer than the original Neo4j.

VII. Conclusions

We proposed GeminiSketch, a data structure for summarizing temporal graph streams by solving the problems of query efficiency and accuracy. GeminiSketch separately records temporal edges according to their arrival time and the distribution of vertex degree for the supporting fast temporal graph queries. A new *Rolling-out* strategy is designed to identify expired edges in a directional way from large temporal edges rather than traversing all buckets, achieving rapid elimination of expired edges. We performed both theoretical analysis and a series of experimental tests. The experimental results show that GeminiSketch can achieve a query-friendly and fast summarization for temporal graph stream with accuracy and efficiency. Regarding future work, we will focus on the compression of temporal graph streams by adopting the idea of GeminiSketch.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments. This work is supported by National Natural Science Foundation of China (No. 62302352, No. U23A20300), Qin Chuang Yuan Fund Program (No. QCYRCXM-2022-355), Key Research Project of Shaanxi Natural Science Foundation (No. 2023-JC-ZD-35), Concept Verification Funding of Hangzhou Institute of Technology of Xidian University (No. GNYZ2024XX007), China 111 Project (No. B16037).

REFERENCES

[1] G. Theodorakis, J. Clarkson, and J. Webber, "Aion: Efficient temporal graph data management." in *Proceedings of the 27th International Conference on Extending Database Technology*, 2024, pp. 501–514.

[2] G. Wang, Y. Zeng, R.-H. Li, H. Qin, X. Shi, Y. Xia, X. Shang, and L. Hong, "Temporal graph cube," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 12, pp. 13 015–13 030, 2023.

[3] L. F. Brito, B. A. Travençolo, and M. K. Albertini, "A review of in-memory space-efficient data structures for temporal graphs," *arXiv preprint arXiv:2204.12468*, 2022.

[4] R. Qiu, Y. Ming, Y. Hong, H. Li, and T. Yang, "Wind-bell Index: Towards ultra-fast edge query for graph databases," in *Proceedings of IEEE 39th International Conference on Data Engineering*, 2023, pp. 2090–2098.

[5] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, "Reachability and time-based path queries in temporal graphs," in *Proceedings of the 32nd International Conference on Data Engineering*, 2016, pp. 145–156.

[6] Y. Matsunobu, S. Dong, and H. Lee, "Myrocks: LSM-tree database storage engine serving facebook's social graph," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3217–3230, 2020.

[7] J. Zhang, C. Gao, D. Jin, and Y. Li, "Group-buying recommendation for social e-commerce," in *Proceedings of IEEE 37th International Conference on Data Engineering*, 2021, pp. 1536–1547.

[8] S. Bhatia, M. Wadhwa, K. Kawaguchi, N. Shah, P. S. Yu, and B. Hooi, "Sketch-based anomaly detection in streaming graphs," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 93–104.

[9] M. Simeonovski, G. Pellegrino, C. Rossow, and M. Backes, "Who controls the internet? analyzing global threats using property graph traversals," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 647–656.

[10] J. Jiang, Y. Li, B. He, B. Hooi, J. Chen, and J. K. Z. Kang, "Spade: A real-time fraud detection framework on evolving graphs," *Proceedings of the VLDB Endowment*, vol. 16, no. 3, pp. 461–469, 2022.

[11] X. Huang, Y. Yang, Y. Wang, C. Wang, Z. Zhang, J. Xu, L. Chen, and M. Vazirgiannis, "Dgraph: A large-scale financial dataset for graph anomaly detection," *Advances in Neural Information Processing Systems*, vol. 35, pp. 22 765–22 777, 2022.

[12] B. Bui Xuan, A. Ferreira, and A. Jarry, "Computing shortest, fastest, and foremost journeys in dynamic networks," *International Journal of Foundations of Computer Science*, vol. 14, no. 02, pp. 267–285, 2003.

[13] D. Caro, M. A. Rodríguez, and N. R. Brisaboa, "Data structures for temporal graphs based on compact sequence representations," *Information Systems*, vol. 51, pp. 1–26, 2015.

[14] N. R. Brisaboa *et al.*, "Using compressed suffix-arrays for a compact representation of temporal-graphs," *Information Sciences*, vol. 465, pp. 459–483, 2018.

[15] D. Caro *et al.*, "Compressed kd-tree for temporal graphs," *Knowledge and Information Systems*, vol. 49, no. 2, pp. 553–595, 2016.

[16] M. Steinbauer and G. Anderst-Kotsis, "Dynamograph: a distributed system for large-scale, temporal graph processing, its implementation and first observations," in *Proceedings of the 25th International Conference Companion on World Wide Web*, 2016, pp. 861–866.

[17] M. Romero, N. Brisaboa, and M. Rodríguez, "The SMO-index: a succinct moving object structure for timestamp and interval queries," in *Proceedings of ACM Sigspatial International Conference on Advances in Geographic Information Systems*, 2012, pp. 498–501.

[18] S. Zang, S. Han, P. Yuan, X. Shi, and H. Jin, "Hyperbit: A temporal graph store for fast answering queries," *Data & Knowledge Engineering*, vol. 144, p. 102128, 2023.

[19] M. Massri, Z. Miklos, P. Raipin, and P. Meye, "Clock-G: A temporal graph management system with space-efficient storage technique," in *Proceedings of IEEE 38th International Conference on Data Engineering*, 2022, pp. 2263–2276.

[20] "Wechat statistics," https://expandedramblings.com/index.php/wechat-statistics/, 2024.

[21] Y. Liu, T. Safavi, A. Dighe, and D. Koutra, "Graph summarization methods and applications: A survey," *ACM computing surveys*, vol. 51, no. 3, pp. 1–34, 2018.

[22] N. Shabani, J. Wu, A. Beheshti, Q. Z. Sheng, J. Foo, V. Haghighi, A. Hanif, and M. Shahabikargar, "A comprehensive survey on graph summarization with graph neural networks," *IEEE Transactions on Artificial Intelligence*, vol. 5, no. 8, pp. 3780–3800, 2024.

[23] M. Besta and T. Hoefler, "Survey and taxonomy of lossless graph compression and space-efficient graph representations," *arXiv preprint arXiv:1806.01799*, 2018.

[24] P. Zhao, C. C. Aggarwal, and M. Wang, "gSketch: On query estimation in graph streams," *Proceedings of the VLDB Endowment*, vol. 5, no. 3, 2011.

[25] N. Tang, Q. Chen, and P. Mitra, "Graph stream summarization: From big bang to big crunch," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1481–1496.

[26] M. S. Hassan, B. Ribeiro, and W. G. Aref, "SBG-sketch: A self-balanced sketch for labeled-graph stream summarization," in *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*, 2018, pp. 1–12.

[27] J. Ko, Y. Kook, and K. Shin, "Incremental lossless graph summarization," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 317–327.

[28] X. Zhu, X. Huang, B. Choi, and J. Xu, "Top-k graph summarization on hierarchical dags," in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, 2020, pp. 1903–1912.

[29] M. Chen, R. Zhou, H. Chen, and H. Jin, "Scube: Efficient summarization for skewed graph streams," in *Proceedings of IEEE 42nd International Conference on Distributed Computing Systems*, 2022, pp. 100–110.

[30] X. Gou, L. Zou, C. Zhao, and T. Yang, "Graph stream sketch: Summarizing graph streams with high speed and accuracy," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 6, pp. 5901–5914, 2022.

[31] Z. Jiang, H. Chen, and H. Jin, "Auxo: A scalable and efficient graph stream summarization structure," *Proceedings of the VLDB Endowment*, vol. 16, no. 6, pp. 1386–1398, 2023.

[32] Y. Wang, H. Chen, H. Chen, and H. Jin, "Sliding-ITeM: An adaptive-size graph stream summarization structure based on sliding windows," *Data Science and Engineering*, pp. 1–27, 2025.

[33] M. Chen, R. Zhou, H. Chen, J. Xiao, H. Jin, and B. Li, "Horae: A graph stream summarization structure for efficient temporal range query," in *Proceedings of 2022 IEEE 38th International Conference on Data Engineering*, 2022, pp. 2792–2804.

[34] S. Ali, M. Ahmad, M. A. Beg, I. U. Khan, S. Faizullah, and M. A. Khan, "Ssag: Summarization and sparsification of attributed graphs," *ACM Transactions on Knowledge Discovery from Data*, vol. 18, no. 6, pp. 1–22, 2024.

[35] J. Guo, B. Chen, K. Yang, T. Yang, Z. Liu, Q. Yin, S. Wang, Y. Wu, X. Wang, B. Cui *et al.*, "Higgs: Hierarchy-guided graph stream summarization," in *Proceedings of IEEE 41st International Conference on Data Engineering*, 2025.

[36] X. Zhao, X. Xie, and C. S. Jensen, "Higgs: Hierarchy-guided graph stream summarization," in *Proceedings of IEEE 41st International Conference on Data Engineering*, 2025.

[37] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen, "Immortalgraph: A system for storage and analysis of temporal graphs," *ACM Transactions on Storage*, vol. 11, no. 3, pp. 1–34, 2015.

[38] S. Ma, R. Hu, L. Wang, X. Lin, and J. Huai, "Fast computation of dense temporal subgraphs," in *Proceedings of IEEE 33rd International Conference on Data Engineering*, 2017, pp. 361–372.

[39] Y. Jia, Z. Gu, Z. Jiang, C. Gao, and J. Yang, "Persistent graph stream summarization for real-time graph analytics," *World Wide Web*, vol. 26, no. 5, pp. 2647–2667, 2023.

[40] 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.

[41] "The source code of geminisketch." https://.

[42] "Sx-stackoverflow," https://snap.stanford.edu/data/index.html.

[43] "wiki-talk temporal network," https://snap.stanford.edu/data/wiki-talk-temporal.html.

[44] "Reddit hyperlink network," https://snap.stanford.edu/data/soc-RedditHyperlinks.html.

[45] "Super user temporal network," https://snap.stanford.edu/data/sx-superuser.html.

[46] "Neo4j. website [online]. https://neo4j.com/," website [online]. https://neo4j.com/.