Dolha - an Efficient and Exact Data Structure for Streaming Graphs

Fan Zhang
Peking University
Beijing, China, 100080
zhangfanau@pku.edu.cn

Lei Zou
Peking University,
Beijing, China, 100080
zoulei@pku.edu.cn

Li Zeng
Peking University
Beijing, China, 100080
li.zeng@pku.edu.cn

Xiangyang Gou
Peking University
Beijing, China, 100080
gxy1995@pku.edu.cn

ABSTRACT

A streaming graph is a graph formed by a sequence of incoming edges with time stamps. Unlike static graphs, the streaming graph is highly dynamic and time related. In the real world, the high volume and velocity streaming graphs such as internet traffic data, social network communication data and financial transfer data are bringing challenges to the classic graph data structures. We present a new data structure: double orthogonal list in hash table (Dolha) which is a high speed and high memory efficiency graph structure applicable to streaming graph. Dolha has constant time cost for single edge and near linear space cost that we can contain billions of edges information in memory size and process an incoming edge in nanoseconds. Dolha also has linear time cost for neighborhood queries, which allow it to support most algorithms in graphs without extra cost. We also present a persistent structure based on Dolha that has the ability to handle the sliding window update and time related queries.

PVLDB Reference Format:
Fan Zhang, Lei Zou, Li Zeng, Xiangyang Gou. Dolha - an Efficient and Exact Data Structure for Streaming Graphs. *PVLDB*, xx(xxx): xxxx-yyyy, 2019.
DOI: https:doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

In the real world, billions of relations and communications are created every day. A large ISP needs to deal about 10^9 packets of network traffic data per hour per router [1]; 100 million users log on Twitter with around 500 million tweets per day [2]; In worldwide, the total number of sent/received emails are more than 200 billion per day [3]. Those relations are coming and fading away like the tides and mining knowledge from the highly dynamic graph data is as difficult like capturing the certain wave of the sea. To handle this situation, we need a graph data structure that has high memory efficiency to contain the enormous amount of data and high speed to seize every nanosecond of the stream.

Proceedings of the VLDB Endowment, Vol. xx, No. xxx
ISSN 2150-8097.
DOI: https:doi.org/10.14778/xxxxxxx.xxxxxxx

There have been several prior arts in streaming graph summarization like TCM [4] and and specific queries like TRIST [5]. However, there are some complicated situations that these existing work did not cover. To illustrate our problem in this paper, we first give some motivation examples as follows:

Use Case 1: Network traffic. The network traffic is a typical kind of streaming graphs. Each IP address indicates one vertex and the communication between two IPs indicates an edge. Along with the data packets sending and receiving, the graph changes rapidly. To monitor this network, we need to run queries on this streaming graph. For example, to detect the suspects of cyber-attack, we want to know how many data packets each IP sends or receives and how many KBs data each edge carries. This problem is defined as vertex query and edge query and could be solved by the graph summarization system [4] in $O(1)$ time cost. However, if we need more structure-aware query answers, such as ”who are the receivers of given IP?”, ”who are the 2-hop neighbors of this IP?” and ”how many IPs that this IP could reach?”, the existing graph summarization techniques (such as TCM [4]) cannot provide accurate query answers. In some applications, an *exact* data structure is desirable for streaming graphs rather than probabilistic data structure.

Use Case 2: Social network. In a social network graph, a user is considered as one vertex and the relations are the edges from this user. One of the most common queries is triangle counting and there are many algorithms to deal with this problem. But existing solutions are designed specifically for triangle counting [5] and so are some continuous subgraph matching systems [6] and circle detecting systems [7] over streaming graphs. If we want to run different kinds of dynamic graph analysis, we have to maintain multiple streaming systems that are costly on both space and time. An elegant solution is one uniform system that could support most graph analysis algorithms on streaming graphs.

Usually, an edge in streaming graph is received with a timestamp indicating the edge arrive time. Some applications need to figure out **historical information** or **time constrains** based on these time stamps, however few systems support these *time-related* graph queries for historical information. Here are two examples:

Use Case 3: Financial transaction. For example, a bank has a streaming graph system to monitor last seven days' money transactions. Each customer is recorded as one vertex, and each money tracer is recorded as one edge. On Friday, the bank receives a notice from another branch that a few suspicious transfers are made on Tuesday between 10am and 4pm from this bank.To find these suspicious transfers, the bank needs to run some pattern match on the time constrained transfers. In this case, we need a streaming graph system not only supports *last snapshot*-based queries but also enable *time-related* queries to figure out historical information.

Use case 4: Fraud detection. The same bank from Case 3 receives another report from police. The report has a list of suspicious accounts that may involves credit card fraud and the money transfer pattern they use. The bank needs to find when such pattern appeared in the transaction record among those accounts. Figure 1 shows an example of credit card fraud pattern. In this case, we have the bank's account ID, merchant account ID and a list of suspicious accounts ID that had transactions with the merchant account. Consider these IDs as vertex and the transactions as edges, we could construct a set of query graphs. We need to locate the occurrence time when these query graphs appear in the streaming transaction graph, then we check inward and outward neighbors of these suspicious accounts near that time and find other criminal group members.

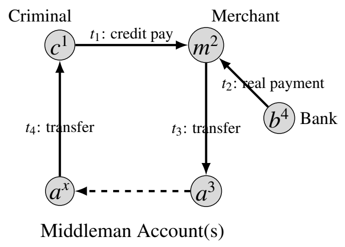


Figure 1: Credit card fraud in transactions (Taken from [7])

Motivated by above use cases, an efficient streaming grap structure should satisfy the following requirements:

- To enable efficient graph computing, the space cost of the data structure should be small enough to fit into main memory;
- For the enormous amount of data and the high-frequency updating, the data structure must have $O(1)$ time cost to handle one incoming edge processing;
- The data structure should support many kinds of graph algorithms rather than designed for one specific graph algorithm;
- The data structure should also support time-related queries for historical information.

In the literature, there exist some streaming graph data structures. Generally speaking, they are classified into two categories: general streaming graph data structure and a data structure designed for some specific graph algorithms. General streaming graph structures are designed to preserve the whole structure of streaming graphs, thus, they can support most of graph algorithms like BFS, DFS, reachability query and subgraph matching by using neighbor search primitives. Most of these kind of structures are based on hash map associated with some classical graph data structures such as adjacency matrix and adjacency list. GraphStream Project [8] is based on adjacency list associated with hash map. The basic idea of this structure is to map the vertex IDs into a hash table. Each cell of vertex hash table stores the vertex ID and the incoming/outgoing links. TCM [4] and gMatrix [9] propose to combine hash map with adjacency matrix. Different from [8], TCM and gMatrix are approximate data structures that inherit query errors due to hash conflicts. There are also some other streaming graph data structures that support a single *specific* graph algorithm, such as HyperANF [10] for t-hop neighbor and distance query, the Single-Sink DAG [11] for pattern matching and TRIEST [5] for triangle counting.

Table 1 lists the space cost of different general streaming graph data structure together with the time complexity to handle edge insertion and edge/1-hop queries. GraphStream's edge insertion time is $O(d)$, which depends on the maximum vertex degree. In many scale-free network data, the maximum vertex degree is often very large. Thus, GraphStream is not suitable for high speed streaming graph. TCM and gMatrix have the square space cost that prevents them to be used in large graphs. On the contrary, our proposed approach (called Dolha) in this paper fits all requirements for streaming graphs. Generally, Dolha is the combination of the orthogonal list with hash techniques. The orthogonal list builds two single linked lists of the outgoing and incoming edges for each vertex and store the first items of two list in vertex cell. On the other hand, the hash table is commonly used for streaming data structure to achieve amortized $O(1)$ time look up, such as bloom filter [12] and count-min [13]. The combination of orthogonal list and hash table is an promising option to achieve our goal. Based on this idea, we present a new exact streaming graph structure: *d*ouble *o*rthogonal *l*ist in *ha*sh table (Dolha).

Table 1: General Streaming Graph Structures

	Adjacency List	**Adjacency Matrix**	**Orthogonal List**
+Hash	**GraphStream [8]**	**TCM [4]**	**Dolha**
Space Cost	$O(\|E\| \log \|V\|)$	$O(\|V\|^2)$	$O(\|E\| \log \|E\|)$
Time Cost per Edge	$O(\log d)$	$O(1)$	$O(1)$
Edge Query	$O(\log d)$	$O(1)$	$O(1)$
1-hop Neighbor Query	$O(d)$	$O(\|V\|)$	$O(d)$

Our Contributions: Table 1 shows the comparison among the three general streaming graph structures. In this paper:

1. We design an effective data structure (Dolha) for streaming graphs with $O(|E| \log |E|)$ space cost and $O(1)$ time cost for a single edge operation. Compared with existing data structures, Dolha is more suitable in the context of high speed straming graph data.
2. The Dolha data structure can answer many kinds of queries over streaming graphs, among which Dolha support the query primitive edge query in $O(1)$ time and 1-hop neighbor queries in $O(d)$ time.
3. We present a variant of Dolha, Dolha persistent that supports sliding window and time related queries in linear time cost.
4. Extensive experiments over both real and synthetic datasets confirm the superiority of Dolha over the state-of-the-arts.

2. RELATED WORK

Among the existing studies, we categorize the structures into two classes: general streaming graph structures and streaming graph algorithms structures.

2.1 General Streaming Graph Structures

General streaming graph structures are designed to preserve the data of graph stream and maintain the graph connection information at the same time. A general streaming graph structures could support most of graph algorithms like BFS, DFS, reachability query and subgraph matching by using neighbor search primitives. Most of these kind of structures are based on hash map associated with basic graph data structure like adjacency matrix and adjacency list. There are two different general streaming graph structures: exact structure and approximation structure.

Exact Structures: Graph Stream Project [8] is an exact graph stream processing system which is implanted by Java. Graph Stream Project is based on adjacency list associated with hash map and it supports most of graph algorithms. The basic idea of this structure is to map the vertex IDs into a hash table. Each cell of vertex hash table stores the vertex ID and the incoming / outgoing links.

Adjacency list needs $O(|E| \log |V|)$ space and $O(|V| + |E|)$ time for traversal. However, to locate an edge, we need to go through the neighbor lists pf both in and out vertices which indicates $O(|E|)$ time cost in some extreme situations. Even we put the neighbor list into a sorted list, it still costs $O(\log d)$ time (d is the average degree of vertices) for each edge look up.

Figure 2 shows an example of adjacency list in hash table. We use hash function $H(*)$ to map the 6 vertices into 6 cells vertex hash table and each cell has 2 sorted list to store the outgoing and incoming neighbors of the vertex. i.e., $H(v_2) = 0$ and cell 0 stores the vertex ID v_2, the outgoing list $\{4 = H(v_3), 5 = H(v_5)\}$ and incoming list $\{2 = H(v_1)\}$. The adjacency list stores the exact information of the graph stream but cost $O(d)$ for each edge insertion.

Vertex Index	0		1		2		3		4		5	
Vertex ID	v_2		v_6		v_1		v_4		v_3		v_5	
	out	in	out	in	out	in	out	in	out	in	out	in
	4	2	2	2	0	1	1	2	3	0	1	0
	5			3	1	3	2	4				2
				5	3							
					5							

Figure 2: Example of adjacency list in hash table

Approximation Structures: Another solution for the structure of streaming graph is adjacency matrix in hash table. We could hash the vertices into a hash table and using a pair of vertices indexes as coordinates to construct an adjacency matrix. Vertex query in hash table is $O(1)$ time cost and so is edge look-up in the matrix. From the view of time cost, adjacency matrix in hash table is efficient but $O(|V|^2)$ space cost is a drawback. In the real world, graphs are usually sparse and we could not afford to spend 2.5 quadrillion on a 50 million vertices graph. There is a compromise formula that we compress the vertices into $O(\sqrt{|E|})$ size or even smaller hash table to reduce the space cost up to $O(|E|)$. But with the high compress ratio, its only suite for a graph summarization system, like TCM [4], gMatrix [9].

Figure 3 shows an example of adjacency matrix in hash table. We use hash function $H(*)$ to map the 6 vertices into 3 cells hash table and use the table index to build a 3×3 matrix. In the 9 cells of the matrix, we store the weights of 11 edges. i.e., $H(v_1) = 1$ and $H(v_2) = 0$, the matrix table cell $(1, 0)$ indicates the edge $\overrightarrow{v_1 v_2}$. However, the cell $(1, 0)$ also indicates the edge $\overrightarrow{v_1, v_6}$ and $\overrightarrow{v_5, v_6}$ since the hash collision. If we do outgoing neighbor query for v_2, the result is $\{v_5, v_1, v_2, v_3\}$ and the correct answer is $\{v_5, v_3\}$. In this case, if we want the exact result, the matrix size is 6×6 which is much larger than the edge size 11.

2.2 Specific Streaming Graph Structures

Vertex Index	0	1	2
Vertex Label	v_2 ,v_6	v_5 ,v_1	v_3 ,v_4

	0	1	2
0	0	3	1
1	1	1	1
2	1	2	1

Figure 3: Example of adjacency matrix in hash table

Unlike the general structure, there are some data structures designed for specific algorithms on graph stream. For example, HyperANF [10] is an approximation system for t-hop neighbor and distance query; the Single-Sink DAG [11] is for pattern matching on large dynamic graph; TRIST [5] is sampling system for triangle counting in streaming graph; and there are some connectivity and spanners structures showed in Graph stream survey [14]. These systems could only support the designed algorithms and become incapable or unacceptable on other graph queries.

Time constrained continuous subgraph search over streaming graphs [6] is a rare and the latest research work that considers the time as query parameter. This paper proposed an a kind of query that requires not only the structure matching but also the time order matching. Figure 4 shows an example of time constrained subgraph query. In this query, each edge of query graph has a time-stamp constrain ϵ. A matching subgraph means the subgraph is an isomorphism of query graph and the time-stamps are following the given order.

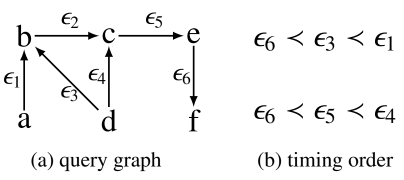

$\epsilon_6 \prec \epsilon_3 \prec \epsilon_1$
$\epsilon_6 \prec \epsilon_5 \prec \epsilon_4$
(a) query graph
(b) timing order

Figure 4: Running example query Q (Taken from [6])

3. PROBLEM DEFINITION

Definition 1 (Streaming Graph). *A streaming graph $\mathcal{G}$ is a directed graph formed by a continuous and time-evolving sequence of edges $\{\sigma_1, \sigma_2, ...\sigma_x\}$. Each edge σ_i from vertex u_i to v_i is arriving at time t_i with weight w_i, denoted as $\sigma_i(\overrightarrow{u_i v_i}, t_i, w_i)$, $i = 1, ..., x$.*

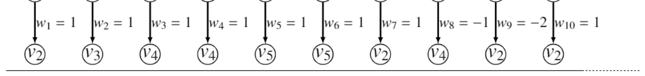


Figure 5: Streaming Graph S

Generally, there are two models of streaming graphs in the literature. One is only to care the latest snapshot structure, where the latest snapshot is the superposition of all coming edges to the latest time point. The other model records the historical information of the streaming graphs. The two models are formally defined in Definitions 2 and 4, respectively. In this paper, we propose a uniform data structure (called *Dolha*) to support both of them.

Definition 2 (Snapshot & Latest Snapshot Structure). *An edge $\overrightarrow{uv}$ may appear in $\mathcal{G}$ multiple times with different weights at different time stamps. Each occurrence of $\overrightarrow{uv}$ is denoted as $\sigma^j(\overrightarrow{uv}, t^j, w^j)$, $j = 1, .., n$. The total weight of edge $\overrightarrow{uv}$ at snapshot t is the weight sum of all occurrences before (and including) time point t, denoted as*

$$W^t(\overrightarrow{uv}) = \sum\nolimits_{t^j \le t} w^j.$$

where $\sigma(\overrightarrow{uv}, t^j, w^j)$ appears in streaming graph $\mathcal{G}$.

For a streaming graph $\mathcal{G}$, the corresponding snapshot at time point t (denoted as $\mathcal{G}_t$) is a set of edges that has positive total weight at time t:

$$\mathcal{G}_t = \{(\overrightarrow{uv}) \in \mathcal{G} \mid W^t(\overrightarrow{uv}) > 0\}.$$

When t is the current time point, $\mathcal{G}_t$ denotes the the latest snapshot structure *of $\mathcal{G}$.*

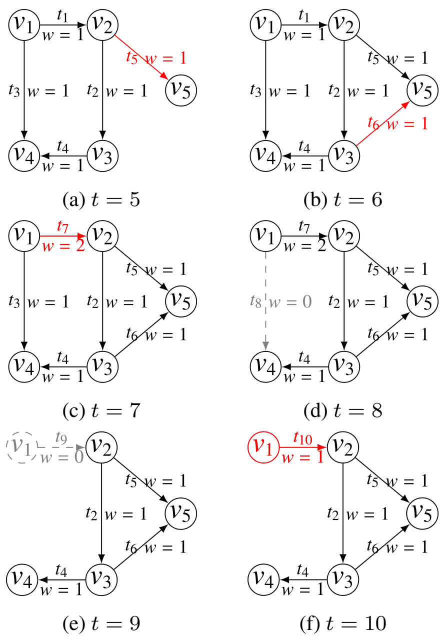


Figure 6: snapshot $\mathcal{G}_5$ to snapshot $\mathcal{G}_{10}$ of streaming graph $\mathcal{G}$

An example of streaming graph $\mathcal{G}$ is shown in Figure 5. Figure 6 shows the snapshots of $\mathcal{G}$ from t_7 to t_{10}. In Figure 6c, total edge weight $\overrightarrow{v_1v_2}$ is updated from $W^1(\overrightarrow{v_1v_2}) = 1$ (at time t_1) to $W^7(\overrightarrow{v_1v_2}) = 2$ (at time t_7). In Figure 6d, edge $\overrightarrow{v_1v_4}$ receives a negative weight update. Since the weight of $\overrightarrow{v_1v_4}$ is 0 after update, it means that it is deleted from the snapshot $\mathcal{G}_8$ at time t_8. In Figure 6e, the deletion of edge $\overrightarrow{v_1v_2}$ causes the deletion of vertex v_1 in $\mathcal{G}_9$ and v_1 is added into $\mathcal{G}_{10}$ again because the new edge $\overrightarrow{v_1v_2}$ incoming at t_{10}.

In some applications, we need to record the historical information of streaming graphs, such as fraud detection example (Use Case 4) in Section 1. Thus, we also consider the sliding window-based model.

Definition 3 (Sliding Window). *Let t_1 be the starting time of a streaming graph $\mathcal{G}$ and w be the window length. In every update, the window would slide θ and $\theta < w$. $D^i_{w,\theta}(\mathcal{G})$ contains all edges in the i-th sliding window, denoted as:*

$$D^i_{w,\theta}(\mathcal{G}) = \{(\overrightarrow{uv}, t, w)|$$

$$(\overrightarrow{uv}, t, w) \in \mathcal{G}, t_0 + (i-1) \times \theta \le t \le t_0 + (i-1) \times \theta + w\}.$$

[15]

In Figure 16, the window size $w = 7$ and each step the window slides $\theta = 3$ edges. Figure 16 illustrates the first and the second sliding window, where the left-most three edges expired in the second window.

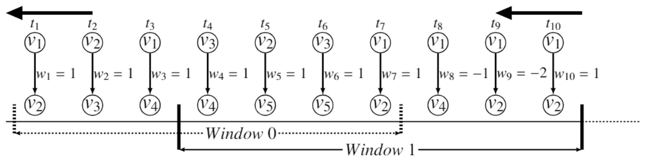


Figure 7: Sliding window update on streaming graph

Definition 4 (Window Based Persistent Structure). *Given a streaming graph $\mathcal{G}$, the* ***Window Based Persistent Structure*** *("persistent structure" for short) is a graph formed by all the unexpired edges in the current time window. Each edge is associated with the time stamps denoting the arriving times of the edge. An edge may have multiple time stamps due to the multiple occurrences.*

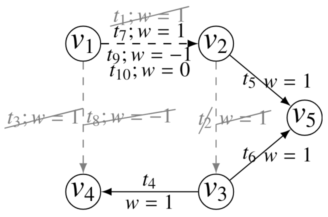


Figure 8: Window based persistent structure

In a snapshot streaming graph structure, only the latest snapshot is recorded and the historical information is overwritten. For example, a snapshot structure only stores the snapshot $\mathcal{G}_{10}$ at last time point t_{10} in Figure 6f. The update process of the streaming graph is overwritten.

Assume that the second time window (Window 1) is the current window. Figure 8 shows how the persistent structure stores the streaming graph. Edge $\overrightarrow{v_1v_2}$ is associated with three time points (t_7, t_9 and t_{10}) that are all in the current time window. Although edge $\overrightarrow{v_1v_2}$ also occurs at time t_1, it is expired in this time window. The gray edges denotes all expired edges, such as $\overrightarrow{v_1v_4}$ and $\overrightarrow{v_2v_3}$.

Definition 5 (Streaming graph query primitives). *We define 4 query primitives for streaming graph $\mathcal{G}$ and most of the graph algorithms such as DFS, BFS, reachability query and subgraph matching are based on these query primitives:*

1. ***Edge Query:*** *Given the a pair of vertices IDs (u, v), return the weight or time stamp of the edge $\overrightarrow{uv}$. If the edge doesnt exist, return $\{null\}$.*

2. ***Vertex Query:*** *Given the a vertex IDs u, return the incoming or outgoing weight of u. If the vertex does not exist, return $\{null\}$.*

3. ***1-hop Successor Query:*** *Given the a vertex IDs u, return a set of vertices that u could reach in 1-hop. If there is no such vertex, return $\{null\}$.*

4. ***1-hop Precursor Query:*** *Given the a vertex IDs u, return a set of vertices that could reach u in 1-hop. If there is no such vertex, return $\{null\}$.*

The query primitives are slightly different in two structures. If we query edge $\overrightarrow{v_1v_2}$ in snapshot structure at $\mathcal{G}_{10}$, the result is the last updated edge information : $(\overrightarrow{v_1v_2}, t_{10}, 1)$. If we query edge $\overrightarrow{v_1v_2}$ in persistent structure at $\mathcal{G}_{10}$ showing in Figure 8, the result is a list of unexpired edges: $(\overrightarrow{v_1v_2}, t_7, 0)$, $(\overrightarrow{v_1v_2}, t_9, -1)$, $(\overrightarrow{v_1v_2}, t_{10}, 1)$. The same difference applies to 1-hop successor query and precursor query. If we query the successor of v_1 at t_{10}, the snapshot structure will give the answer v_2. But the persistent structure will return a set of answers: (v_2, t_7), (v_2, t_9), (v_2, t_{10}).

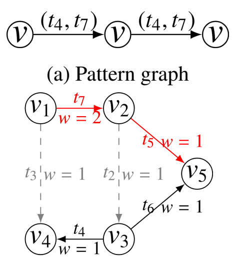
Based on the persistent structure query primitives, we define a new type of queries on streaming graph named *time related query* that considers the time stamps as query parameters. In this paper, we adopt two kinds of time related queries: time constrained pattern query is to find the match subgraph in a given time period; structure constrained time query is to find the time periods that given subgraph appears in $\mathcal{G}$.

Definition 6 (Time Constrained Pattern Query). *A pattern graph is a triple $P = (V(P), E(P), L)$, where $V(P)$ is a set of vertices in P, $E(P)$ is a set of directed edges, L is a function that assigns a label for each vertex in $V(P)$. Given a pattern graph P and a time period (t, t') and $t < t'$, $\mathcal{G}$ is a time constrained pattern match of P if and only if there exists a bijective function F from $V(P)$ to $V(g)$ such that the following conditions hold:*

1. ***Structure Constraint (Isomorphism)***
   - $\forall u \in V(P), L(u) = L(F(u))$.
   - $\overrightarrow{uv} \in E(P) \Leftrightarrow \overrightarrow{F(u)F(v)} \in E(g)$.

2. ***Time Period Constraint***
   - $\forall \overrightarrow{uv} \in E(P), t \leq t_{\overrightarrow{uv}} \leq t'$. *[6]*

In this paper, the problem is to find all the time constrained pattern matches of given P over $\mathcal{G}_{t'}$ which is the snapshot of $\mathcal{G}$ at time t'.



(a) Pattern graph

(b) Pattern match

Figure 9: Time constrained pattern query

Figure 9 shows an example of time constrained pattern query. In Figure 9a, a pattern graph is given which queries all the 2-hop connected structures. The edges of pattern graph have a time constrain that only the edges with the time stamp between (t_4, t_7) are considered as match candidates. Figure 9b is the snapshot $\mathcal{G}_7$ of $\mathcal{G}$ at time t_7. Edge $\overrightarrow{v_1v_4}$ and $\overrightarrow{v_2v_3}$ are discarded since the time stamps are out of time constrain. Edge set $\{(\overrightarrow{v_1v_2})(\overrightarrow{v_2v_5})\}$ is the only matching subgraph for the given pattern on $\mathcal{G}$.

Definition 7 (Structure Constrained Time Query). *A query graph Q is a sequence of directed edges $\{q_1, q_2, ..., q_m\}$ and T is a set of time pairs $\{(t_1, t_1^{'}), ..., (t_n, t_n^{'})\}$. Given a pattern graph Q, a structure constrained time match T is that Q is the subgraph of every snapshot of $\mathcal{G}$ during any time period $(t_i, t_i^{'})$ in T.*

$$\forall t, t_i \leq t \leq t_i^{'}, Q \in G_t.$$

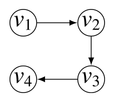


Figure 10: Structure constrained time query

Figure 10 gives an example of structure constrained time query edge set $\{\overrightarrow{v_1v_2}, \overrightarrow{v_2v_3}, \overrightarrow{v_3v_4}\}$ is given. On $\mathcal{G}$, the query graph is the subgraph of every snapshot from $\mathcal{G}_4$ to $\mathcal{G}_8$ until deletion of $\overrightarrow{v_1v_2}$ on $\mathcal{G}_9$. In $\mathcal{G}_{10}$, the query graph is matching again since the new arriving $\overrightarrow{v_1v_2}$. The query result of Figure 10 is $\{(t_4, t_7), (t_{10}, t_{10})\}$.

4. DOLHA - DOUBLE ORTHOGONAL LIST IN HASH TABLE

Table 2: Notations

Notation	Definition and Description
G_s / G_t	Streaming graph / Snapshot at time point t
D_s / D_p	Dolha snapshot / Dolha persisdent
$\overrightarrow{uv}$	The directed edge from vertex u to v
Doll	Doulble orthogonal linked list
O	Outgoing Doll
I	Incoming Doll
T	Time travel linked list
w	Edge weight
t	Edge time stamp
$H(*)$	Hash value of $*$
$V(*)$	Vertex table index of $*$
$E(*)$	Edge table index of $*$
$E^*_A()$	First item's edge table index of link $*$
$E^*_\Omega()$	Last item's edge table index of link $*$
$E^*_\Omega()$	Last item's edge table index of link $*$
$E^*_N()$	Next item's edge table index of link $*$
$E^*_P()$	Previous item's edge table index of link $*$
$*^{-/+}$	Previous/next item of $*$

In order to handle high speed streaming graph data, we propose the data structure—called Double Orthogonal List in Hash Table (*Dolha* for short)—in this paper. Essentially, Dolha is the combination of double orthogonal linked list with hash tables. A *d*ouble *o*rthogonal *l*inked *l*ist (*Doll* for short) is a classical data structure to store a graph, in which each edge $\overrightarrow{uv}$ in graph $\mathcal{G}$ is both in the double linked list of all the outgoing edges from vertex u: $\{\overrightarrow{uv_A}, ...\overrightarrow{uv_\Omega}\}$ denotes as *outgoing Doll* and in the double linked list of all the incoming edges to vertex v: $\{\overrightarrow{u_Av}, ...\overrightarrow{u_\Omega v}\}$ denotes as *incoming Doll*. Vertex u has two pointers to the first item v_A and last item v_Ω of outgoing Doll. Vertex v has two pointers to the first item u_A and last item u_Ω of incoming Doll. For example, Figure 11 illustrates an example of Doll.

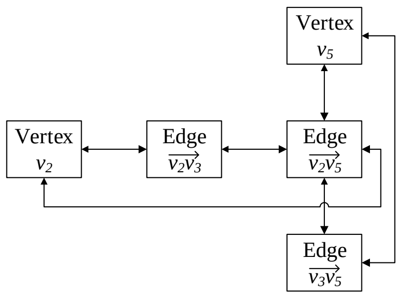

Vertex v_5
Vertex v_2
Edge $\overrightarrow{v_2v_3}$
Edge $\overrightarrow{v_2v_5}$
Edge $\overrightarrow{v_3v_5}$

Figure 11: Example of Doll

4.1 Dolha Snapshot Data Structure

Given a graph $\mathcal{G}$, the Dolha structure contains of four key-value tables. Before that, we assume that each vertex u (and edge $\overrightarrow{uv}$) is hashed to a hash value $H(u)$ (and $H(\overrightarrow{uv})$). For example, we use hash function $H(*)$ to map the vertices and edges:

- $H(v_1) = 1, H(v_2) = 2, H(v_3) = 0, H(v_4) = 1, H(v_5) = 3$
- $H(\overrightarrow{v_1v_2}) = 1, H(\overrightarrow{v_2v_3}) = 0, H(\overrightarrow{v_1v_4}) = 4, H(\overrightarrow{v_3v_4}) = 2, H(\overrightarrow{v_2v_5}) = 4, H(\overrightarrow{v_3v_5}) = 3$

Vertex Hash Table: Dolha creates $m_v(m_v \geq |V|)$ size vertex hash table and uses function $H(*)$ map the vertex ID u to vertex hash table index $H(u)$. Due to the hash collision, there could be a list of vertices with same hash table index. In each table cell, Dolha stores the vertex table index of the first vertex on collision list.

Table 3 is an example of vertex hash table. We use $H(v_1) = 1$ as hash index to locate the vertex table index 0 and find the v_1's details in vertex table cell 0. The vertex v_4 has the same hash value as v_1 which means the hash collision occurs. We use hash value 1 to find the first vertex v_1 on the collision list then we can find the next item v_4's vertex table index 3 in v_1's vertex table cell.

Vertex Table V**:** Dolha creates $m_v(m_v \geq |V|)$ size vertex table and one empty cell variable denoted as ϕ_V. Initially, $\phi_V = 0$. We denote the vertex table index for new coming vertex u as $V(u)$. Let $V(u) = \phi_V$ and increase ϕ_V by 1. In each vertex table cell, Dolha stores the vertex ID, the outgoing weight sum $w_O(u)$ and incoming weight sum $w_I(u)$, the head and tail edge table index for outgoing Doll , the head and tail edge table index for incoming Doll and the vertex table index of the next vertex on collision list .

Table 4 shows the vertex table of $\mathcal{G}_5$ in Figure 6. Out/In w indicates the outgoing and incoming weights of the vertex. O is the edge table index of first and last items of outgoing Doll and I is the edge table index of first and last items of incoming Doll. H is the next vertex on the collision list. The vertices are given indexes incrementally ordered by first arriving time. $\phi_V = 5$ means vertex table is full. If more vertices arrive, we can create a new vertex table and begin with index 5 as the extension of existing vertex table.

Edge Hash Table: Edge hash table: Dolha creates $m_e(m_e \geq |E|)$ size vertex hash table and uses function $H(*)$ map the outgoing vertex ID u plus incoming vertex ID v of edge $\overrightarrow{uv}$ to edge hash table index $H(\overrightarrow{uv})$. Same as the vertex hash table, Dolha stores the edge table index of the first edge on collision list .

In Table 5, we have the same method as vertex hash table to deal with hash collision. $\overrightarrow{v_1v_4}$ has the same hash value 4 as $\overrightarrow{v_2v_5}$. In cell 4, we can find $\overrightarrow{v_1v_4}$'s edge table index 2 then find $\overrightarrow{v_2v_5}$'s edge table index.

Edge Table E: Dolha creates $m_e(m_e \geq |E|)$ size vertex table and one empty cell flag denoted as ϕ_E. Initially, $\phi_E = 0$. We denote the vertex table index for new coming edge $\overrightarrow{uv}$ as $E(\overrightarrow{uv})$. Let $E(\overrightarrow{uv}) = \phi_E$ and increase ϕ_E by 1. In each edge table cell, Dolha stores the vertex table indexes $V(u)$ and $V(v)$, the weight $w(\overrightarrow{uv})$, the time stamp $t(\overrightarrow{uv})$, the previous and next edge table index for outgoing Doll , the previous and next edge table index for incoming Doll and the edge table index of the next edge on collision list
.

Table 6 shows the edge table of $\mathcal{G}_5$ in Figure 6. w is the weight and t is the time stamp. Vertex index indicates the outgoing and incoming vertices of the edge. O is the edge table index of next and previous items of outgoing Doll and I is the edge table index of next and previous items of incoming Doll. H is the next edge on the collision list.

Table 3: Vertex hash table of $\mathcal{G}_5$

Hash index	**0**	**1**	**2**	**3**	**4**
Vertex table index	2	0	1	4	/

Table 4: Vertex table of $\mathcal{G}_5$

Index	**0**		**1**		**2**		**3**		**4**	
Vertex ID	v_1		v_2		v_3		v_4		v_5	
Out/In w	2	0	2	1	1	1	0	2	0	1
O	0	2	1	4	3	3	/	/	/	/
I	/	/	0	0	1	1	2	3	4	4
H	3		/		/		/		/	

$\phi_V = 5$

Table 5: Edge hash table of $\mathcal{G}_5$

Hash index	**0**	**1**	**2**	**3**	**4**	**5**
Edge table index	1	0	3	/	2	/

Table 6: Edge table of $\mathcal{G}_5$

Index	**0**		**1**		**2**		**3**		**4**		**5**	
w	**1**		**1**		**1**		**1**		**1**		**/**	
t	**1**		**2**		**3**		**4**		**5**		**/**	
Vertex index	0	1	1	2	0	3	2	3	1	4	/	/
O	/	2	/	4	0	/	/	/	1	/	/	/
I	/	/	/	/	/	3	2	/	/	/	/	/
H	/		/		4		/		/		/	

$\phi_E = 5$

4.2 Dolha Snapshot Construction

When an edge $(\overrightarrow{uv}; t; w)$ comes:

- Map the edge $\overrightarrow{uv}$ into edge hash table cell $H(\overrightarrow{uv})$.
- If $H(\overrightarrow{uv})$ is empty, $\overrightarrow{uv}$ does not exist in D_s. If $H(\overrightarrow{uv})$ is not empty, traverse the collision list of cell $H(\overrightarrow{uv})$ in edge hash table. If find $\overrightarrow{uv}$, $\overrightarrow{uv}$ exists; if not, $\overrightarrow{uv}$ does not exist.

There are two possible operations:
If $\overrightarrow{uv}$ does not exist in D_s:

- Add $\overrightarrow{uv}$ into into edge table cell $E(\overrightarrow{uv})$ and the collision list of $H(\overrightarrow{uv})$.

- Map the vertices u,v into vertex hash table $H(u)$,$H(v)$.
- If $H(u)$ is empty, add ID u into vertex table cell $V(u)$. If $H(u)$ is not empty, traverse the collision list of cell $H(u)$ in vertex hash table. If find match ID, then we update vertex table $V(u)$ of u; if not, add u into vertex table cell $V(u)$ and collision list of $H(u)$.
- Do the same operation for v.
- Add $\overrightarrow{uv}$ into the end of outgoing Doll of u and incoming Doll of v.

If $\overrightarrow{uv}$ exists in D_s:

- Set $t(\overrightarrow{uv}) = t$ and $w(\overrightarrow{uv}) = w(\overrightarrow{uv}) + w$.
- Delete $\overrightarrow{uv}$ from outgoing Doll of u and incoming Doll of v
- If $\overrightarrow{uv}$ has positive weight after this update:
- Add $\overrightarrow{uv}$ into the end of outgoing and incoming Dolls.
- if $\overrightarrow{uv}$ has zero or negative weight after this update:
- Delete $\overrightarrow{uv}$ from edge table.
- If there is not any item in both Doll of u or v, delete u or v.

For example, at time 6, edge $\overrightarrow{v_3v_5}$ is received. We use $H(v_3v_5) = 3$ to get the edge hash table index and find edge $\overrightarrow{v_3v_5}$ is a new edge. We write the empty cell index 5 of edge table into hash table and check the two vertices by using vertex hash table. We locate the $V(2)$ for v_3 and $V(4)$ for v_5 on vertex table and get the last item of outgoing Doll $E(3)$ and the last item of incoming Doll $E(4)$. We update the both last items of outgoing and incoming Doll to 5 then move to edge table. We update the next item of outgoing Doll to 5 in $E(3)$ and update the next item of incoming Doll to 5 in $E(4)$. Finally, we write w, t, $(2, 4)$, $(3, /)$ and $(4, /)$ into $E(5)$.

At time 7, edge $\overrightarrow{v_1v_2}$ comes and it is already on the edge table. We first update the w and t at $E(0)$ and remove $\overrightarrow{v_1v_2}$ from both of the Dolls then add it to the end of Dolls.

At time 8, edge $\overrightarrow{v_1v_4}$ carries negative weight and w is 0 after the update. We move $E(2)$ from the outgoing and incoming doll and update the associated indexes, then we empty the cell 2 of edge table and put the index 2 into empty edge cell list. At time 9, edge $\overrightarrow{v_1v_2}$ is deleted and v_1 has no out or in edges. We empty cell 0 of vertex table and put the index 0 into empty vertex cell list.

4.3 Time and Space Cost

4.3.1 Time Cost

Algorithm 1 shows how Dolha process one incoming edge.

From line 3 to 14, we maintain the edge hash table to check the existence of incoming edge $\overrightarrow{uv}$. According to [16], if we hash n items into a hash table of size n, the expected maximum list length is $O(\log n/\log\log n)$. In the experiment, more than 99% collision list is less than $\log n/\log\log n$, more than 90% collision list is shorter than 5. Hash table could achieve amortized $O(1)$ time cost for 1 item insert, delete and update which is much faster than sorted table. This step costs $O(1)$ time.

If $\overrightarrow{uv}$ is a new edge, from line 16 to 22, we maintain the vertex hash table to check the existence of two vertices u and v. In this step, we do two hash table look up and it costs $O(1)$ time. From line 23 to 29, we write $\overrightarrow{uv}$ into edge table then add it into the end of outgoing and incoming Dolls. The time complexity of this step is same as insertion on double linked list which is $O(1)$.

Algorithm 1: Dolha snapshot edge processing

```
   Input: Streaming graph $\mathcal{G}$
   Output: Dolha snapshot structure of $\mathcal{G}$
1  for each incoming edge $(\overrightarrow{uv}; t; w)$ of $\mathcal{G}$ do
2      Check existence of $\overrightarrow{uv}$:
3      Map $\overrightarrow{uv}$ into $H(\overrightarrow{uv})$.
4      if $H(\overrightarrow{uv})$ is null then
5          $\overrightarrow{uv}$ does not exist
6      else
7          Traverse the collision list from $E^H_A(\overrightarrow{uv})$.
8          if reach null and no match for $\overrightarrow{uv}$ then
9              $\overrightarrow{uv}$ does not exist
10         else
11             $\overrightarrow{uv}$ exists
12     if $\overrightarrow{uv}$ does not exist then
13         Update collision list of $\overrightarrow{uv}$:
14         if $H(\overrightarrow{uv})$ is empty then
15             Let $E^H_A(\overrightarrow{uv}) = E(\overrightarrow{uv})$
16         else
17             Let $E^H_N(\overrightarrow{uv}^-) = E(\overrightarrow{uv})$
18         Check existence of $u$:
19         Map the vertices $u$ into $H(u)$
20         if $H(u)$ is null then
21             Add $u$ into vertex table $V(u)$ and let $V^H_A(u) = V(u)$
22         else
23             Traverse the collision list from $E^H_A(u)$.
24             if reach null and no match for $u$ then
25                 Add $u$ into vertex table $V(u)$ and let
                     $V^H_N(u^-) = V(u)$
26         Do the same operation for $v$ same as $u$
27         Add $\overrightarrow{uv}$ into edge table $E(\overrightarrow{uv})$
28         Add $\overrightarrow{uv}$ into outgoing Doll:
29         if both $E^O_A(u)$ and $E^O_\Omega(u)$ are null then
30             Let $E^O_A(u) = E(\overrightarrow{uv})$ and $E^O_\Omega(u) = E(\overrightarrow{uv})$
31         if neither $E^O_A(u)$ nor $E^O_\Omega(u)$ is null then
32             Let $E^O(\overrightarrow{uv}^-) = E^O_\Omega(u)$ and $E^O_N(\overrightarrow{uv}^-) = E(\overrightarrow{uv})$ and
                 $E^O_P(\overrightarrow{uv}) = E^O(\overrightarrow{uv}^-)$ and $E^O_\Omega(u) = E(\overrightarrow{uv})$
33         Add $\overrightarrow{uv}$ into incoming Doll same as outgoing Doll
34     if $\overrightarrow{uv}$ exists then
35         Let $w(\overrightarrow{uv}) += w$ and $t(\overrightarrow{uv}) = t$
36         Delete $\overrightarrow{uv}$ from outgoing Doll:
37         if $\overrightarrow{uv}$ is the first item of outgoing Doll then
38             Let $E^O_A(\overrightarrow{uv}) = E^O_N(\overrightarrow{uv})$ and $E^O_P(\overrightarrow{uv}^+) = null$
39         if $\overrightarrow{uv}$ is the last item of outgoing Doll then
40             Let $E^O_\Omega(\overrightarrow{uv}) = E^O_P(\overrightarrow{uv})$ and $E^O_N(\overrightarrow{uv}^-) = null$
41         else
42             Let $E^O_N(\overrightarrow{uv}^-) = E^O_N(\overrightarrow{uv})$ and $E^O_P(\overrightarrow{uv}^+) = E^O_P(\overrightarrow{uv})$
43         Delete $\overrightarrow{uv}$ from incoming Doll same as outgoing Doll
44         if $w(\overrightarrow{uv}) > 0$ then
45             Add $E(\overrightarrow{uv})$ into the end of outgoing Doll and incoming
                 Doll
46         else
47             Delete $\overrightarrow{uv}$
48             Delete $E(\overrightarrow{uv})$ and flag $E(\overrightarrow{uv})$ as empty cell
49             if there is no item on outgoing Doll or incoming Doll of
                 $u$ then
50                 Delete $V(u)$ and flag $V(u)$ as empty cell
51             if there is no item on outgoing Doll or incoming Doll of
                 $v$ then
52                 Delete $V(v)$ and flag $V(v)$ as empty cell
```

If $\overrightarrow{uv}$ exists, from line 31 to 38, we update the weight and time stamp of $\overrightarrow{uv}$ then delete it from outgoing and incoming Dolls. This step costs the same time as deletion on double linked list which is also $O(1)$. From line 39 to 40, if updated weight is positive, we add the $\overrightarrow{uv}$ to the end of both two Dolls which costs $O(1)$. If the updated weight is zero or negative, we delete $\overrightarrow{uv}$ completely then delete u and v if they have 0 in and out degrees. Line 41 to 46 shows the deletions and this step also costs $O(1)$.

Overall, for each incoming edge processing, the time complexity of Dolha is $O(1)$.

4.3.2 Space Cost

Dolha snapshot structure needs one $|V|$ cells vertex hash table, one $|V|$ cells vertex table, one $|E|$ cells edge hash table and one $|E|$ cells edge table. Dolha also needs a $\log |V|$ bits integer for one vertex index and $\log |E|$ bits for one edge index.

Vertex hash table: Each cell only stores one vertex index. It costs $\log |V| \times |V|$ space.

Edge hash table: Each cell only stores one edge index. It costs $\log |E| \times |E|$ space.

Vertex table: Each cell stores vertex ID, in and out weights one $\log |V|$ bits vertex index for collision list, four $\log |E|$ bits edge indexes for Dolls. It costs $(\log |V| + 4 \times \log |E|) \times |V|$ space.

Edge table: Each cell stores weight, time stamp, one $\log |E|$ bits edge index for collision list, two $\log |V|$ bits vertex index for in and out vertices, four $\log |E|$ bits edge indexes for Dolls. It costs $(2 \times \log |V| + 5 \times \log |E|) \times |E|$ space.

In total, Dolha needs $(2 \times \log |V| + 4 \times \log |E|) \times |V| + (2 \times \log |V| + 5 \times \log |E|) \times |E|$ bits for the data structure. Since usually $|V| \ll |E|$, the space cost of Dolha snapshot structure is $O(|E| \log |E|)$.

5. DOLHA PERSISTENT STRUCTURE

5.1 Dolha Persistent Data Structure

Using Dolha, We could construct a persistent structure D_p and D_p contains any snapshot's information of $\mathcal{G}$. D_p has the same structure as D_s except the time travel list.

Definition 8 (Time Travel List). *An edge $\overrightarrow{uv}$ may appear in streaming graph S multiple times with different time stamp. Time travel list T is a single linked list that links all the edges $\overrightarrow{uv}$ which share same outgoing and incoming vertices. In T, each edge has an index points to its previous appearance in the stream.*

D_p also has four index-value tables. The vertex hash table, vertex table and edge hash table are same as D_s. In each cell of edge table, D_p has a extra value which indicates the previous item on the time travel list.

5.2 Dolha Persistent Construction

5.2.1 Incoming Edge Processing

When an edge $\sigma(\overrightarrow{uv}; t; w)$ comes:

- Check the existence of $\overrightarrow{uv}$ same as Dolha snapshot.

If $\overrightarrow{uv}$ does not exist in D_p:

- The operation is exact same as Dolha snapshot.

If $\overrightarrow{uv}$ exists in D_p:

- Use edge hash table to find the existing edge table index $E(\sigma')$ of $\overrightarrow{uv}$.
- Insert edge σ as new edge into edge table and set the time travel list index as $E(\sigma')$.
- Update the edge table index of $\overrightarrow{uv}$ on edge hash collision list.

Algorithm 2: Dolha persistent edge processing

```
Input: Streaming graph $\mathcal{G}$
Output: Dolha persistent structure of $\mathcal{G}$
1  for each incoming edge $\sigma(\overrightarrow{uv}; t; w)$ of $\mathcal{G}$ do
2      Check existence of $\overrightarrow{uv}$:
3      if $\overrightarrow{uv}$ does not exist then
4          Insert $\overrightarrow{uv}$
5      if $\overrightarrow{uv}$ exists in cell $E(\sigma')$ then
6          Insert $E(\sigma)$ as new edge and let $w(\sigma) = w(\sigma) + w(\sigma')$
7          Let $E_P^T(\sigma) = E(\sigma')$
8      if value of $H(\overrightarrow{uv})$ in edge hash table is $null$ then
9          Let $E_A^H(\overrightarrow{uv}) = E(\sigma)$
10     else
11         Let $E_N^H(\overrightarrow{uv}^-) = E(\sigma)$
```

Table 7: Edge hash table of Window 0

Hash index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Edge table Index	1	/	3	/	5	4	/	2	6	/

Table 8: Edge table of Window 0

Index	**0**		**1**		**2**		**3**		**4**		**5**		**6**		**7**		**8**		**9**	
w	**1**		**1**		**1**		**1**		**1**		**1**		**2**		**/**		**/**		**/**	
t	**1**		**2**		**3**		**4**		**5**		**6**		**7**		**/**		**/**		**/**	
V	0	1	1	2	0	3	2	3	1	4	2	4	0	1	/	/	/	/	/	/
O	/	2	/	4	0	6	/	5	1	/	3	/	2	7	/	/	/	/	/	/
I	/	/	/	6	/	3	2	7	/	5	4	/	1	8	/	/	/	/	/	/
H	/		/		/		/		/		/		/		/		/		/	
T	/		/		/		/		/		/		0		/		/		/	

Table 9: Edge table of Window 1

Index	**0**		**1**		**2**		**3**		**4**		**5**		**6**		**7**		**8**		**9**	
w	**/**		**/**		**/**		**1**		**1**		**1**		**1**		**-1**		**0**		**/**	
t	**/**		**/**		**/**		**4**		**5**		**6**		**7**		**9**		**10**		**/**	
V	/	/	/	/	/	/	2	3	1	4	2	4	0	1	0	1	0	1	/	/
O	/	/	/	/	/	/	/	5	/	/	3	/	/	7	6	8	7	/	/	/
I	/	/	/	/	/	/	/	/	/	5	4	/	/	7	6	8	7	/	/	/
H	/		/		/		/		/		/		/		/		/		/	
T	/		/		/		/		/		/		/		6		7		/	

Table 7 and 8 show the Dolha persistent's edge hash table and edge table of $\mathcal{G}$ in Window 0. The vertex hash table and vertex table of Dolha persistent are similar like Dolha snapshot and so is the new edge coming. But for edge $\overrightarrow{v_1 v_2}$ update at time 6, we add the update as new edge into $E(6)$ and update the edge hash table to latest update. By using the time travel list, all the updates of $\overrightarrow{v_1 v_2}$ are linked.

5.2.2 Sliding Window Update

When the window slides the ith step, we have the start time $t_s = t_0 + (i-2) \times \theta$ and end time $t_e = t_0 + (i-1) \times \theta$ of expired edges which need to delete from edge table. Since the edge table is naturally ordered by time, we can find the last expired edge denote as $E(\sigma_e)$ at t_e in $O(\log S)$ time. By using edge hash table, we can find the latest update of $E(\sigma_\Omega)$ and traversal back by the time travel list. For each $E(\sigma_n)(e < n \leq \Omega)$ on time travel list, let $w_n = w_n - w_e$. If each $w_n \leq 0$, delete all the $E(\sigma_n)$. Then delete each $E(\sigma_m)(0 < m \leq e)$ on time travel list. Do the same operation for the edges from t_e to t_s. For every deleted edge, if it is the first or last item of Doll, update the associated cell in vertex table and set the index to $null$. If all the Doll indexes are $null$ in that vertex cell, delete the vertex and flag the cell as empty.

As shown in Figure 16, when window slides from 0 to 1 means the edges before t_4 will expire. First, we can binary search the edge table to locate the first unexpired edge index 3 since the table is sorted by time stamp. Then we start to delete the expired edges from cell 3. We use the hash table to check if there are unexpired updates for the expired edges. For example, $\overrightarrow{v_1v_2}$ has unexpired update at time 7, so we minus the expired weight at cell 6.

Table 9 shows the edge table of Dolha persistent at Window 1. The first 3 expired edges have been deleted. At time 8, $\overrightarrow{v_1v_4}$ with negative weight arrives, but there is no positive $\overrightarrow{v_1v_4}$ in this window. In this case, we won't save $\overrightarrow{v_1v_4}$. At time 9 and 10, $\overrightarrow{v_1v_2}$ has negative or zero weights, but $\overrightarrow{v_1v_4}$ has positive weight at time 7, so we keep the record and link them with time travel linked list.

Space Recycle: Due to the chronological ordered edge table, the expired edges are always continuous and in the head of the unexpired edges. We could always recycle the space from expired edges which means we won't need infinite space to save the continuous streaming but only need the maximum number of edges in each window. For instance, in table 9, we can re-use the cell from 0 to 1 for next window update and we have enough space as long as no more than 9 edges in 1 window.

5.3 Time and Space Cost

The time cost of Dolha persistent is hash table cost, Doll cost and time travel list cost. For each incoming edge, the hash table cost and Doll cost are $O(1)$ as we discussed in Dolha snapshot and the time travel list cost is also $O(1)$ same as insertion on single linked list. Overall, the time cost for one edge processing is $O(1)$.

To store all the information of streaming S, Dolha persistent structure needs one $|V|$ cells vertex hash table, one $|V|$ cells vertex table, one $|S|$ cells edge hash table and one $|S|$ cells edge table. In total, Dolha needs $(2\times\log|V|+4\times\log|S|)\times|V|+(2\times\log|V|+5\times\log|S|)\times|S|$ bits plus $\log|S|\times|S|$ for time travel list. The space cost of Dolha persistent structure is $O(|S|\log|S|)$.

6. ALGORITHMS ON DOLHA

In this section, we discuss how to perform the graph algorithms on both Dolha snapshot structure and persistent structure.

6.1 Algorithms on Dolha Snapshot

6.1.1 Query Primitives

Dolha snapshot structure supports all the 4 graph query primitives.

Edge Query: Given a pair of vertices IDs (u, v), to query the weight and time stamp of edge $(\overrightarrow{uv})$ is same as the existence checking of $(\overrightarrow{uv})$ in insertion operation. By using edge hash table, we can find $E(\overrightarrow{uv})$ on edge table and return w and t. As we proved before, the time cost of hash table checking is amortized $O(1)$.

Vertex Query: Similar as edge query, by using vertex hash table, we can locate given vertex u on vertex table in $O(1)$ time and return the query result.

1-hop Successor Query and 1-hop Precursor Query: Given a vertex ID u, Dolha first perform vertex query to find $V(u)$ in $O(1)$ time. Then we have the head edge index $E_A^O(u)$ of outgoing Doll. From $E(\sigma) = E_A^O(u)$, we can use $E_N^O(\sigma)$ to acquire all edges on outgoing Doll iteratively and add the incoming vertex indexes of these edges into set $\{V(v)\}$. The IDs of $\{V(v)\}$ can be found in vertex table and returned as the results of 1-hop successor query. The 1-hop precursor query is similar as successor query but use incoming Doll instead. The time cost of Doll iteration depends on the outgoing or incoming degree d of given u. The total time cost of 1-hop successor query or 1-hop precursor query is $O(d)$.

Chronological Doll: In Dolha structure, we maintain the Doll in chronological order. The result list of 1-hop successor query or 1-hop precursor query is sorted by the time stamps. The chronological Doll could reduce the search space in some time related queries. For example, in Figure 4, we have a candidate edge $(\overrightarrow{uv}; t)$ that matches $(\overrightarrow{dc}; \epsilon_4)$ and look for the candidate edges of $(\overrightarrow{ce}; \epsilon_5)$. Since the timing order constrain $\epsilon_5 \prec \epsilon_4$, we first check the time stamp of first edge on v's outgoing Doll in $O(1)$ time. If the time stamp is equal or larger than t, it means there is no match for $(\overrightarrow{ce}; \epsilon_5)$. If the time stamp is less than t, we can search from the first edge on v's outgoing doll until equal or larger the time stamp than t.

6.1.2 Directed Triangle Finding

By using the 4 graph query primitives, most graph algorithms could run on Dolha. The 1-hop successor query and 1-hop precursor query associated with edge query could support all the BFS or DFS based algorithms like reachability query, tree parsing, shortest path query, subgraph matching and triangle finding. For example, the triangle finding is a common graph query on streaming graph.

To query the directed triangle on Dolha, we can use the edge iterator method. During the Dolha snapshot construction, we can add one out degree counter and one in degree counter for each vertex. For each edge $(\overrightarrow{uv})$ incoming edge, get the minimal candidate set $\{j\}$ between v's successor set and u's precursor set. Then check each j in set $\{j\}$ that if there is $(\overrightarrow{ju})$ or $(\overrightarrow{vj})$ existing in edge table by using edge query. The set of all existing $(\overrightarrow{uv}, \overrightarrow{vj}, \overrightarrow{ju})$ is the query result.According to [17], the time complexity of triangle finding on whole graph is $O(\sum_{\overrightarrow{uv}\in E}\min\{d_{in}(u), d_{out}(v)\})$, so the time cost is $O(\min\{d_{in}(u), d_{out}(v)\})$ for each edge update.

Algorithm 3: Continuous directed triangle finding on Dolha snapshot

```
Input: Dolha snapshot structure of $\mathcal{G}$ with out and in degree counter
Input: Streaming Graph $\mathcal{G}$
Output: Directed triangles in $\mathcal{G}$
1 for each new coming edge $\overrightarrow{uv}$ of $\mathcal{G}$ do
2     if in degree of $u \leq$ out degree of $v$ then
3         for each vertex $j$ in $u$'s precursor set do
4             if $\overrightarrow{vj}$ exsits in edge table then
5                 Put $(\overrightarrow{uv}, \overrightarrow{ju}, \overrightarrow{vj})$ into result set
6     else
7         for each vertex $j$ in $v$'s successor set do
8             if $\overrightarrow{ju}$ exsits in edge table then
9                 Put $(\overrightarrow{uv}, \overrightarrow{ju}, \overrightarrow{vj})$ into result set
```

6.2 Algorithms on Dolha Persistent

6.2.1 Query Primitives

Dolha persistent structure also supports all the 4 graph query primitives both on the latest snapshot and persistent perspective of $\mathcal{G}$:

Edge Query: Given a pair of vertices IDs (u, v), the latest update of edge $\overrightarrow{uv}$ could be found by using edge hash table. Once find the latest update of edge $\overrightarrow{uv}$, we could use time travel list to retrieve all the updates of $\overrightarrow{uv}$ in current window.

Vertex Query: The vertex query on Dolha persistent is exactly same as snapshot structure.

1-hop Successor Query and 1-hop Precursor Query: Given a vertex ID u, the outgoing or incoming Doll of u may contain duplicates of edges. To query the successor of u on Dolha persistent, it's better from the last item of outgoing Doll $E_\Omega^O(u)$ which is definitely the latest outgoing edge from u. Let $E(\overrightarrow{uv}) = E_\Omega^O(u)$, we add v to the result set and use the time travel link of $\overrightarrow{uv}$ to flag all the previous update records of $\overrightarrow{uv}$. Then we traversal the outgoing doll and do the same operation for each unflagged edge as

$\overrightarrow{uv}$. 1-hop precursor query is same as successor query but using the incoming Doll. The two lists are sorted by time naturally.

6.2.2 *Time Related Queries*

Time Constrained Pattern Query: Given time period (t, t'), the essential part of time constrained pattern query is to find the all the edges with time stamp $(t \leq t_{\overrightarrow{uv}} \leq t')$ on snapshot G'_t. The chronological edge table allows us to locate the first edge $E(\sigma_t)$ at time t and the last edge $E(\sigma'_t)$ at time t' in $O(\log S)$ time. Then we can run Algorithm 4 to construct the adjacency list of the candidate subgraph of time constrained pattern query. We also could construct a Dolha snapshot structure to store the candidate subgraph by using Algorithm 5. The time cost of candidate subgraph construction is $O(\log S + S')$ and the space cost is $O(S')$ (S' is the incoming edge number of (t, t')). We can run any isomorphism algorithm on the candidate subgraph structure to get the final query result.

Algorithm 4: Adjacency list construction for candidate subgraph of time constrained pattern query

```
Input: edges between $\sigma_t$ and $\sigma'_t$ in edge table
Input: Dolha persistent structure of $\mathcal{G}$
Output: Adjacency list of candidate subgraph
1  for each edge $\sigma(\overrightarrow{uv}; t; w)$ from $E(\sigma'_t)$ to $E(\sigma_t)$ do
2      if $flag! = 3$ then
3          for each edge on the time travel list after $\sigma$ do
4              Let $flag = 3$
5          if $flag == 2$ or $flag == 0$ then
6              Put $u$ into candidate vertex set
7              for each edge $\sigma_O$ on outgoing Doll of $u$ do
8                  if $flag! = 3$ then
9                      for each edge on the time travel list after $\sigma_O$ do
10                         Let $flag = 3$
11                     Let $flag+ = 1$
12                     Put the incoming vertex of $\sigma_O$ into the
                         outgoing neighbor list of $u$
13         if $flag == 1$ or $flag == 0$ then
14             Put $v$ into candidate vertex set
15             for each edge $\sigma_I$ on incoming Doll of $v$ do
16                 if $flag! = 3$ then
17                     for each edge on the time travel list after $\sigma_I$ do
18                         Let $flag = 3$
19                     Let $flag+ = 2$
20                     Put the outgoing vertex of $\sigma_I$ into the incoming
                         neighbor list of $v$
```

Algorithm 5: Dolha snapshot construction for candidate subgraph of time constrained pattern query

```
Input: edges between $\sigma_t$ and $\sigma'_t$ in edge table
Input: Dolha persistent structure of $\mathcal{G}$
Output: Dolha snapshot of candidate subgraph
1  Construct a Dolha snapshot structure $D'_t$ with vertex and edge size
   $|E(\sigma'_t) - E(\sigma_t)|$ for each edge $\sigma(\overrightarrow{uv}; t; w)$ from $E(\sigma'_t)$ to $E(\sigma_t)$
   do
2      if $flag! = 1$ then
3          for each edge on the time travel list after $\sigma$ do
4              Let $flag = 1$
5          Insert $\sigma$ into $D'_t$
```

Structure Constrained Time Query: Given a sequence of directed edges $Q\{q_1, q_2, ..., q_m\}$, for each edge q_n in Q, we can use edge hash table to locate the latest update $E(q_n)$ in $\mathcal{G}$ and use time travel list to find the time period set T_n that edge q_n appears. Then we join all the time period sets to find result time period set. The Algorithm 6 shows that the time complexity is $O(m \times p \times \log(m \times p))$ (p is the average number of one edge appearance in S).

Algorithm 6: Structure constrained time query

```
Input: a sequence of directed query edges $Q\{q_1, q_2, ..., q_m\}$
Input: Dolha persistent structure of $\mathcal{G}$
Output: Time period set $T$ that match the query structure
1  Let $t_e = null$ and $t_s = null$
2  Let chronological order set $T_c = \phi$
3  for each edge $q_n$ in $Q$ do
4      Use edge hash table to find the latest update $E(q_n)$
5      for each edge $q^t_n$ on time travel list of $q_n$ from $E(q_n)$ do
6          if $w^t_n > 0$ then
7              Let $t_s = t^t_n$
8              if $t_e == null$ then
9                  Let $t_e = t^t_n$
10         if $w^t_n \leq 0$ then
11             if $t_s! = null$ then
12                 Put $t_s$ flag as $s$ and $t_e$ flag as $e$ into $T_c$
13                 Let $t_e = null$ and $t_s = null$
14 for each item $c_e$ that flagged as e in $T_c$ do
15     if there are $m$ continuous $s$ items on the left of $c$ then
16         Let $c_s =$ the closest left $s$ item
17         Put $(c_s, c_e)$ into $T$
```

7. EXPERIMENTAL EVALUATION

7.1 Experiment Setup

We evaluate Dolha snapshot and Dolha persistent structure separately.

In Dolha snapshot experiment, we compare Dolha snapshot with adjacency matrix in hash table and adjacency list in hash table. Since TCM is based on adjacency matrix in hash table and the java project GraphStream is based on adjacency list in hash table, we believe the comparison to these two general GraphStream structures could reflect the performance of Dolha properly. For the three structures, we first compare the average operation time cost and space cost and then compare the speed of query primitives.

We use the same hash function (MurmurHash) for all the structures and build the same vertex hash table and vertex table for all three structures so they all share the same vertex operation time cost and accuracy. Because the full adjacency matrix is too large, we compress the matrix in certain ratios that costs similar space as Dolha. That makes TCM become an approximation structure and we take account of the relative error.

In Dolha persistent experiment, since there is no similar system for comparison, we build an adjacency list in hash table with an extra time line which stores all the edge update information. We use the adjacency list as baseline method to compare with Dolha persistent on the speed of sliding window update, query primitives and time related queries.

7.1.1 *Dataset*

1. **DBLP [18]:** DBLP dataset contains $1,482,029$ unique authors and $10,615,809$ time-stamped coauthorship edges between authors (about 6 million unique edges). Its a directed graph and we assign each streaming edge with weight 1.

2. **GTGraph [19]:** We use the graph generator toll GTGraph to generate a directed graph. We use the R-MAT model generate a large network with power-law degree distributions and add weight 1 to for each edge and use the system clock to

get the time-stamp. The generated graph contains 30 million vertices and 1 billion stream edges.

3. **Twitter [20]:** We use the Twitter link structure data with 56 million vertices and 2 billion edges as a directed streaming graph and assign weight 1 to each edge.

4. **CAIDA [21]:** CAIDA Internet Anonymized Traces 2015 Dataset obtained from www.caida.org. The network data contains $445,440,480$ communication records (edges) (about 100 million unique edges) concerning $2,601,005$ different IP addresses (vertices).

We use 4 datasets for Dolha snapshot experiment: The DBLP, GTGraph and Twitter are used for Dolha snapshot experiments and DBLP and CAIDA are used for Dolha persistent experiments.

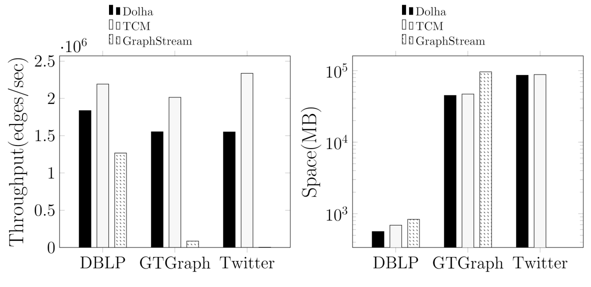
7.1.2 Environment

All experiments are performed on a server with dual 8-core CPUs (Intel Xeon CPU E5-2640 v3 @ 2.60GHz) and 128 GB DRAM memory, running CentOS. All the data structures are implemented in C++.

7.2 Dolha Snapshot Experimental Results

7.2.1 Construction

Firstly, we compare the average processing time cost of stream graph on three structures and the space cost of them. In real world scenario, the insertion, deletion and update operations are usually coming randomly and the average stream processing speed is the key performance indicator of the system and all three operations time costs on Dolha are $O(1)$. So we load the datasets 2 times as insertion and update and set the weight to -3 for last loading as deletion. Then we calculate the average time as the stream processing time cost and present it in the form of operations per second. During the data loading, we record the actual memory consuming when the edges are fully loaded. The results are showing in Figure 18.



(a) Time Efficiency (b) Space Efficiency

Figure 12: Time and space cost for 3 streaming graph structure

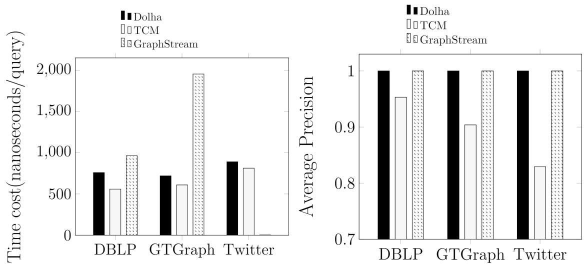
In DBLP dataset, Dolha processing speed reaches $1,837,357$ operations per second which almost same as TCMs $2,192,715$ operations per second and faster than GraphStreams $1,266,815$ operations per second. Since the preset compress ratio, the memory cost of TCM is 690MB which is similar to Dolhas 563MB. The GraphStream costs 833MB which is worse than Dolha. In GTGraph dataset, the performance remains the same. The TCM is the fastest structure with $2,014,768$ operations per second and Dolha is not far behind with $1,552,536$ operations per second. The speed of GraphStream drops significantly to $85,441$ operations per second and the space cost reaches 96GB which is way higher than Dolhas 45GB and TCMs 47GB. In Twitter dataset, the GraphStream runs out memory since the enormous space cost of sorted list maintenance. The performances of Dolha and TCM are steady. Dolha costs 86GB memory and reaches $1,550,197$ operations per second while the TCM costs 88GB and reaches $2,336,785$ operations per second. The time cost results show that Dolha is slightly slower on stream processing speed than the TCM but significantly faster than the GraphStream. Since the TCM is an approximation structure and Dolha is an exact structure, the latency is acceptable. The space cost results show that Dolha could process 2 billion edges stream on less than 90GB memory.

7.2.2 Query Primitives

In this part, we compare the query primitives speed on the three systems. The vertex query, the edge query, 1-hop successor query and 1-hop precursor query are taken into account. The time-related query and sliding window update are not supported by the other two structures and the time costs are depended on the given parameters, so we have not run experiment on these two queries.

Vertex Query: The three structures share the same vertex hash table and vertex table, so the vertex query speeds are same. We run 25 random vertex queries which cost $14,146$ nanoseconds in total. It means the average vertex query is 566 nanoseconds per query.

Edge Query: We run 50 random edge queries for three structures on each dataset. The results show that speed of edge query on Dolha is similar as on TCM with 0 relative error and much faster than GraphStream.



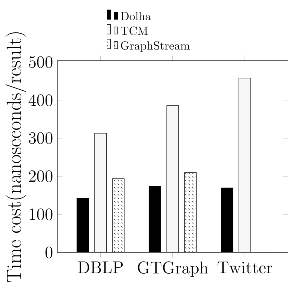
(a) Edge Query Time Cost (b) Edge Query Average Precision

Figure 13: Time cost and average precision for edge query

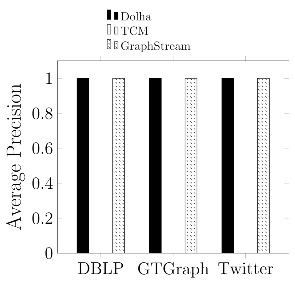
1-hop Successor Query and 1-hop Precursor Query: We randomly choose 25 vertices and run 1-hop successor query and 1-hop precursor query for three structures on each dataset. Since the query speed depends on the size of results set, we calculate the average query speed as nanoseconds per result. The TCM has almost 0 average precision on these queries and slowest query speed. Among the threes structures, Dolha has the best performance with fast query speed and 100% precision.

Compare to the GraphStream, Dolha has great advantages on the average stream processing time cost, space cost, edge query speed, 1-hop successor query and 1-hop precursor query speed. Dolha is slightly slower than the TCM with similar space cost on average stream processing time cost, space cost, edge query speed but faster on 1-hop successor query and 1-hop precursor query. On the other hand, the Dolha is an exact structure and the TCM is an approximation structure.

Directed Triangle Finding: We run continuous directed triangle finding algorithm on DBLP and GTGraph 1 billion date set using Dolha snapshot and GraphStream. For DBLP dataset, Dolha could process 759,866 edge updates per-second and GraphStream only could process 238,095 edge updates per-second. For GTGraph 1

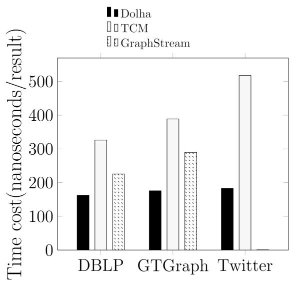


(a) 1-Hop Successor Query Time Cost

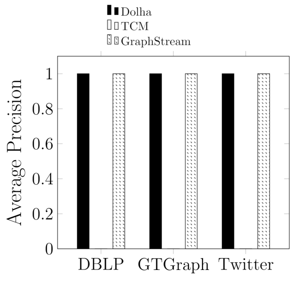


(b) 1-Hop Successor Query Average Precision

Figure 14: Time cost and average precision for 1-hop successor query



(a) 1-Hop Precursor Query Time Cost



(b) 1-Hop Precursor Query Average Precision

Figure 15: Time cost and average precision for 1-hop precursor query

billion date set, Dolha could deal 129,853 throughput edges per-second but GraphStream could only deal less than 10,000 throughput edges per-second.

7.3 Dolha Persistent Experimental Results

7.3.1 Construction and Sliding Window Update

We set window length $= \frac{1}{10}|S|$, slide length $= \frac{1}{5}$ window length as W1 and slide length $= \frac{1}{50}$ window length as W2. Then we load the DBLP and CAIDA dataset with and without sliding window update. Figure 16 shows the through-puts of Dolha persistent and adjacency list plus time-line with and without sliding window update.

On DBLP date set, Dolha persistent reaches $2,008,420$ edges update per second without sliding window update, $1,979,889$ edges update per second in W1 and $1,961,238$ edges update per second in W2. The adjacency list plus time-line only can process $1,120,269$ edges update per second without sliding window update, $893,795$ edges update per second in W1 and $583,367$ edges update per second in W2.

On CAIDA dataset, Dolha persistent reaches $3,969,514$ edges update per second without sliding window update, $3,917,037$ edges update per second in W1 and $3,425,009$ edges update per second in W2. The results are way better than the adjacency list plus time-line's speeds: $761,834$ edges update per second without sliding window update, $676,077$ edges update per second in W1 and $472,953$ edges update per second in W2.

The construction time costs in different window setting on Dolha persistent are similar which means the the size of slide length are insignificant to the edge processing. The outstanding high speed is caused by the high duplicated edge rate on CAIDA dataset. We set the edge hash table same size as edge table, but the unique edge number is only $\frac{1}{4}$ of total stream edge number which reduces the hash collision significantly. And when we process the duplicate edge update, we do not need to check the vertices by using vertex hash table.

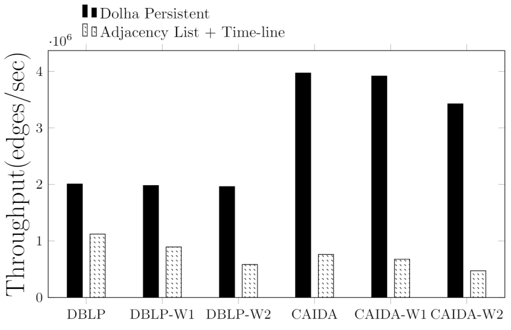


Figure 16: Edge throughput without and with time window update

7.3.2 Query Primitives

The query primitives of DBLP on Dolha persistent are exact the same as Dolha snapshot, we only compare the CAIDA with adjacency list plus time-line.

Vertex Query: The two structures use the same vertex hash table and vertex table. We run 25 random vertex queries and the average vertex query is 605 nanoseconds per query.

Edge Query: We run 50 random edge queries on both data structures. The result shows that Dolha persistent is 5 times faster than adjacency list plus time-line.

1-hop Successor Query and 1-hop Precursor Query: We randomly choose 25 vertices and run 1-hop successor query and 1-hop precursor query on two structures. Dolha persistent is slighly faster than adjacency list plus time-line.

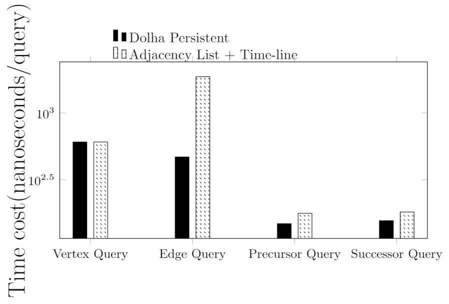
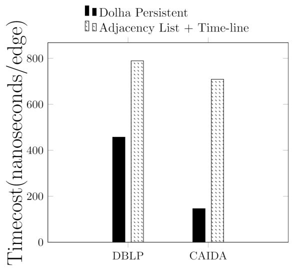


Figure 17: Query primitives on CAIDA

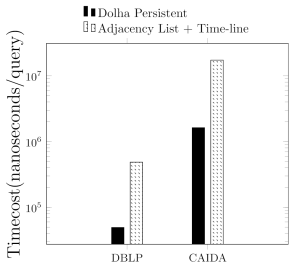
7.3.3 Time Related Queries

Time Constrained Pattern Query: For time constrained pattern query, we randomly choose 3 pairs of time-stamps as time constrain and extract the eligible edges to form a candidate subgraph. Figure 18a shows the average candidate subgraph forming speeds of Dolha persistent and adjacency list plus time-line. In DBLP, we reach 457 nanoseconds per edge to extract the candidate subgraph into a Dolha snapshot and the adjacency list plus time-line can only construct 789 nanoseconds per edge into an adjacency list. In CAIDA, the speed reaches 146 nanoseconds per edge and the adjacency list plus time-line can only process 709 nanoseconds per edge.

Structure Constrained Time Query: To compare structure constrained time query, we randomly choose 5 query edge sets and each set has 5 edges. The average query time of Dolha persistent is $49,378$ nanoseconds per query on DBLP and $1,623,200$ nanoseconds per query on CAIDA. The average query time of adjacency list plus time-line is $486,576$ nanoseconds per query on DBLP and $17,312,871$ nanoseconds per query on CAIDA.



(a) Time Constrained Query



(b) Structure Constrained Time Query

Figure 18: Time related query

8. CONCLUSIONS

We have proposed an exact streaming graph structure Dolha which could maintain high speed and high volume streaming graph in linear time cost and near linear space cost. We have shown that Dolha is a general propose structure that could support the query primitives which are the cornerstone of common graph algorithms. We also present the Dolha persistent structure which could support sliding window update and time related queries. The experiment results have proved that Dolha has better performance than the other streaming graph structures.

9. ADDITIONAL AUTHORS

10. REFERENCES

[1] S. Guha and M. Andrew, "Graph synopses, sketches, and streams: a survey," *PVLDB*, vol. 5, no. 12, pp. 2030–2031, 2012.

[2] "Tweet statistics," http://expandedramblings.com/index.php/march-2013-by-the-numbers-a-few-amazingtwitter-stats/10/.

[3] "Email statistics report, 2015-2019," https://radicati.com/wp/wp-content/uploads/2015/02/Email-Statistics-Report-2015-2019-Executive-Summary.pdf.

[4] N. Tang, Q. Chen, and P. Mitra, "Graph stream summarization: From big bang to big crunch," *SIGMOD*, pp. 1481–1496, 2016.

[5] L. De 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 the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. ACM, 2016.

[6] Y. Li, L. Zou, M. T. Ozsu, and D. Zhao, "Time constrained continuous subgraph search over streaming graphs," https://arxiv.org/pdf/1801.09240.pdf, 2018.

[7] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, "Real-time constrained cycle detection in large dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, 2018.

[8] Y. Pigne, A. Dutot, F. Guinand, and D. Olivier, "Graphstream: A tool for bridging the gap between complex systems and dynamic graphs," *EPNACS*, 2007.

[9] A. Khan and C. C. Aggarwal, "Query-friendly compression of graph streams," *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pp. 130–137, 2016.

[10] P. Boldi, M. Rosa, and S. Vigna, "Hyperanf: approximating the neighbourhood function of very large graphs on a budget," *International world wide web conferences*, pp. 625–634, 2011.

[11] J. Gao, C. Zhou, J. Zhou, and J. X. Yu, "Continuous pattern detection over billion-edge graph using distributed framework," in *Proc. 30th IEEE International Conference on Data Engineering*, 2014, pp. 556–567.

[12] A. Z. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[13] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *latin american symposium on theoretical informatics.*, pp. 29–38, 2004.

[14] A. Mcgregor, "Graph stream algorithms: a survey," *SIGMOD Record*, vol. 43, no. 1, pp. 9–20, 2014.

[15] L. Gao, L. Golab, M. T. Ozsu, and G. Aluc, "Stream watdiv: A streaming rdf benchmark," no. 3, 2018.

[16] C. Stein, S. Drysdale, and K. Borgart, "Probability calculations in hashing," in *Discrete Mathematics for Computer Scientists*. Addison-Wesley; 1st edition, 2010, pp. 245–254.

[17] T. Schank and D. Wagner, "Finding, counting and listing all triangles in large graphs, an experimental study," in *Nikoletseas S.E. (eds) Experimental and Efficient Algorithms. Lecture Notes in Computer Science*, vol. 3503, 2005.

[18] E. Demaine and M. Hajiaghayi, "Bigdnd: Big dynamic network data," http://projects.csail.mit.edu/dnd/DBLP/.

[19] "Gtgraph: A suite of synthetic random graph generators," http://www.cse.psu.edu/~kxm85/software/GTgraph/.

[20] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring User Influence in Twitter: The Million Follower Fallacy."

[21] "Caida internet anonymized traces 2015 dataset," http://www.caida.org/home/.