TgStore: An Efficient Storage System for Large Time-Evolving Graphs

Yongli Cheng, *Member, IEEE*, Yan Ma, Hong Jiang, *Fellow, IEEE*, Lingfang Zeng, *Member, IEEE*, Fang Wang, *Member, IEEE*, Xianghao Xu, *Member, IEEE*, and Yuhang Wu

***Abstract*—Existing graph systems focus mainly on the execution efficiency of the graph analysis tasks, often ignoring the importance and efficiency of time-evolving graph storage. However, to effectively mine the potential application values, an efficient storage system is important for time-evolving graphs whose storage requirement scales with the increasing number of snapshots. Storage cost and snapshot access speed are the two most important performance indicators for a time-evolving graph storage system, which are challenging for designers of such systems because they are conflicting goals. In this article, we address these challenges by proposing an efficient storage scheme for the large time-evolving graphs. We first design a *Snapshot-level Data Deduplication (SLDD)* strategy to eliminate the large number of repeated vertices and edges among the snapshots, and then a *Structure-Changing Graph Representation (SCGR)* to significantly improve the snapshot access speed. We implement an efficient time-evolving graph storage system, TgStore, based on this scheme to effectively store large-scale time-evolving graphs, aiming to efficiently support the time-evolving graph analysis tasks. Experimental results show that TgStore can obtain a high compression ratio of 43.03:1 when storing 100 snapshots of Twitter, while with an average snapshot access speedup of 16×. Efficient storage scheme enables TgStore to efficiently support time-evolving graph algorithms. For example, when executing the Pagerank algorithm on the time-evolving graph of Twitter, TgStore outperforms Graphone, a state-of-the-art time-evolving graph storage system, by 15.9× in algorithm execution speed and 1.45× in memory usage.**

Manuscript received 8 November 2022; revised 12 December 2023; accepted 22 January 2024. Date of publication 14 February 2024; date of current version 14 March 2024. This work was supported in part by the Natural Science Foundation of Fujian Province under Grant 2020J01493, in part by Zhejiang provincial "Ten Thousand Talents Program" under Grant 2021R52007, in part by the Center-initiated Research Project of Zhejiang Lab under Grant 2021DA0AM01, and in part by the Open Project, Program of Wuhan National Laboratory for Optoelectronics under Grant 2022WNLOKF017. Recommended for acceptance by J. Yin. *(Corresponding authors: Lingfang Zeng; Xianghao Xu.)*

Yongli Cheng is with the College of Computer and Data Science, Fuzhou University, Fuzhou 350025, China, also with the Fujian Key Laboratory of Network Computing and Intelligent Information Processing, Fuzhou University, Fuzhou 350025, China, also with the Engineering Research Center of Big Data Intelligence Ministry of Education, Fuzhou 350025, China, and also with the Zhejiang Lab, Hangzhou 311121, China (e-mail: chengyongli@fzu.edu.cn).

Yan Ma and Yuhang Wu are with the College of Computer and Data Science, Fuzhou University, Fuzhou 350025, China (e-mail: 200327075@fzu.edu.cn; 200327087@fzu.edu.cn).

Hong Jiang is with the Department of Computer Science & Engineering, University of Texas at Arlington, Arlington, TX 76019 USA (e-mail: hong.jiang@uta.edu).

Lingfang Zeng is with the Zhejiang Lab, Hangzhou 311121, China (e-mail: zenglf@zhejianglab.com).

Fang Wang is with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: wangfang@hust.edu.cn).

Xianghao Xu is with the School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing 210094, China (e-mail: xianghao@njust.edu.cn).

Digital Object Identifier 10.1109/TBDATA.2024.3366087

***Index Terms*—Time-evolving graph, storage system, data representation, data deduplication.**

I. Introduction

OVER the past decade, many graph systems have been built to support complex graph analysis tasks, such as Pregel [1] from Google. Most of them focus mainly on execution efficiency of the graph analysis tasks, while ignoring the importance and efficiency of graph storage. However, the sudden outbreak of Coronavirus Disease 2019 (COVID-19) has revealed that well-designed time-evolving graph storage systems are also very important and a key to efficient graph computation systems. A graph representing real-world applications usually evolves over time as the vertices and edges change, which can be captured by a series of snapshots, each reflecting the state of the graph for a certain time point. The set of these snapshots of a real-world graph is defined as a time-evolving graph. Specifically, an efficient graph storage system is necessary to store the large number of snapshots obtained frequently and continuously during the evolution of the pandemic. Each snapshot, typically, can be used to model persons and the relationships among them at a certain time point. For example, Guo et al. [2] model time-evolving graph of COVID as follows. The time interval between any two consecutive snapshots is 14 days. In each snapshot, each vertex represents one person. An edge means that two persons are in contact. Vertices are divided into five risk levels as follows. (1) Test positive: infected persons; (2) RL-1: 1-hop neighbors of test positive persons; (3) RL-2: 2-hop neighbors of test positive persons; (4) RL-3: 3-hop neighbors of test positive persons; (5) Safety: risk-free persons. By using the collected and stored data of snapshots, evolution of the pandemic can be tracked in real time, and trend prediction algorithms can also be designed by using the computation results of historical snapshots [3].

More recently, it is encouraging to note that some scholars and major corporations have begun to focus on graph storage systems and their efficeincy, due to the increasing application requirements from industrial projects [4]. However, there is still a hard and long way to go for the technology of graph storage system to reach a maturity level like that of relational database management systems (DBMS), due to some stumbling blocks that need to be overcome and key techniques that require improvement. This paper focuses mainly on the challenges of storage cost and snapshot access speed facing graph storage systems that are designed to support graph analysis.

Storage cost and *snapshot access speed* are the two most important performance indicators when designing a time-evolving graph storage system. Because reducing storage cost requires graph data compression that often renders accessing individual snapshots difficult and time-consuming, they can be conflicting design goals and difficult for designers to stride a good balance between them. That is, on the one hand, time-evolving graph storage techniques, such as graph compression, should be well designed to collect and store the large number of snapshots obtained frequently and continuously from the external business system, with a sufficiently low per-snapshot storage cost. This helps storage systems efficiently support the time-evolving graph analysis task that usually needs to process many snapshots simultaneously. On the other hand, low storage cost usually comes at the expense of low snapshot access speed when adding the new snapshots into the graph storage system and loading the stored snapshots into memory to support analysis tasks.

In this paper, we present an efficient time-evolving graph storage system, called TgStore, that provides an ecosystem to manage the large number of snapshots of a time-evolving graph, with low storage cost and high access speed of snapshots. This efficiency stems from two key designs, that is, the *Snapshot-level Data Deduplication (SLDD)* and the *Structure-Changing Graph Representation (SCGR)*. *SLDD*, as the name implies, is designed to remove duplicate objects (vertex or edge) by first obtaining a union of all snapshots, set (S∪), so that each object (vertex or edge) that would otherwise appear repeatedly in different snapshots occurs only once in set (S∪). This method is inspired by Huo and Tsotras [5] who use snapshot union set to design shortest path algorithm where the snapshot union set is employed to compute the lower limit of the shortest path value among the snapshots. Different from them, we employ snapshot union set to reduce the average per-snapshot storage cost, aiming to design an efficient time-evolving graph storage system. Experimental results show that this method can significantly reduce the storage cost of an average snapshot. The benefit of this method is based on the high snapshot similarity of real-world time-evolving graphs, e.g., only about 0.09% of edges between any two consecutive snapshots of the Weibo dataset[1] are different. The higher snapshot similarity between any two consecutive snapshots means that the difference between them is smaller, resulting in a smaller snapshot union set (S∪) of a time-evolving graph. Each object (vertex or edge) in the snapshot union is associated with a bitmap with bits indicating the object's presence/absence (1/0) in all snapshots. By using these bitmaps, any snapshot can be easily reconstructed based on the presence/absence of an object in the snapshot, without any costly decompression operation, resulting in a high speed of snapshot reconstructing. Like bitmap-based designs for other purposes, the bitmap-based TgStore will be space-efficient [6], [7] where only one bit is needed to represent a repeated object in a snapshot.

However, it is nontrivial to design an efficient data structure for *Snapshot-level Data Deduplication*. The reason is that, comparing with the static graph, the graph structure of the snapshot union set (S∪) of a time-evolving graph is dynamically changing with the added vertices or edges of each newly appended snapshot. To append a new snapshot, the storage method for the structure-changing graph usually needs to execute many data movement operations on disk files [8], resulting in a long appending time.

In order to address this challenge, we design an efficient data structure for the structure-changing S∪ by improving the log-based graph data representation [9], [10]. The log-based approach stores a time-evolving graph by using log structure built on a baseline snapshot, which logs changes to the baseline that come from the external business system. The log records all the deleted and added vertices and edges from the last snapshot to represent a new snapshot, without any data movement operations. However, log-based method has two limitations. First, there are still many repeated vertices and edges in the log since vertices and edges can be removed from one snapshot and added back in one of the subsequent snapshots. Second, it requires a long time to reconstruct a snapshot by using the baseline snapshot and the log segment between the baseline and the targeted snapshot. Like the log-based graph data representation, our method also obtains the new snapshot by logging changes from the external business system, since it is easy for the external business system to record the changes [11]. Unlike the log-based graph data representation, however, we process a log segment differently when appending a new snapshot, as follows. We create an incremental edge data block for the new snapshot, which only stores the new edges that are not included in snapshot union set S∪. We do not store the deleted edges in the new edge data block; instead, for each deleted edge we first lookup its location in the edge data block of the baseline snapshot or one of the incremental edge data blocks, and then append a bit with the value of 0 to the bitmap of this edge. This indicates that this edge is moved from the new snapshot. For each added edge, if it has existed in the old snapshot union set S∪, similar to the case of a deleted edge, a bit with the value of 1 is appended to the bitmap of this edge, indicating that this edge was deleted in one of the previous snapshots but is added back in the current snapshot. Otherwise, the added edge will be added to the new edge data block, where a new bitmap will be created for this brand-new edge with the last bit being set to 1 while all other bits to 0.

The *Structure-Changing Graph Representation* has a high speed of appending snapshots since the difference between any two consecutive snapshots is usually small, needing only a small number of lookups, each requiring one random access. There may be exceptions to this norm. Take the COVID-19 pandemic as an example, a large number of vertices and edges can be removed or deleted from a new snapshot because of the rapid development of the pandemic. To address this problem, we improve our method by traversing the edge data block of the baseline snapshot, the incremental edge data blocks and the new log segment, to execute the task of appending a new snapshot. It is more efficient if the number of the added and removed edges is sufficiently large, since sequential accesses are faster than random accesses in both disk and memory. TgStore adaptively adopts one of the two methods to obtain an optimal speed of

[1][Online]. Available: https://www.aminer.cn/influencelocality

snapshot appending by evaluating the number of the removed and added edges.

TgStore, an efficient storage system for time-evolving graphs, has been designed and implemented in this paper to support single-node and in-memory time-evolving graph analysis tasks [4], [12]. Although we believe that it can also be built on a distributed system [13], [14], it is beyond the scope of this paper and we leave it as a topic of future study.

This paper makes the following contributions:

- *An efficient storage scheme for time-evolving graphs:* we first design *Snapshot-level Data Deduplication (SLDD)* to reduce the per-snapshot storage cost, which also enables high-speed snapshot reconstruction. We then design *Structure-Changing Graph Representation (SCGR)* to address the data movement problem of *SLDD*, resulting in high-speed snapshot appending. SCGR also optimizes the edge storage, further reducing the per-snapshot storage cost.
- *Implementation of TgStore:* we implement TgStore based on the storage scheme, with about 5000 lines of C code. Experimental results show that TgStore has a high compression ratio of 43.03:1 when storing 100 snapshots of Twitter. The reduced storage cost further speeds up average snapshot loading by 16×.

The rest of the paper is structured as follows. Background and motivation are presented in Section II. Section III introduces the system design of TgStore. Experimental evaluations of the TgStore prototype are presented in Section IV. We discuss related work in Section V and conclude the paper in Section VI.

II. Background and Motivation

In this section, we first introduce the time-evolving graphs in Section II-A. We then discuss single-snapshot oriented methods in Section II-B and multiple-snapshots oriented methods in Section II-C. The discussion and analysis motivate us to present TgStore.

A. Time-Evolving Graphs

A time-evolving graph (*TEG*) consists of a series of snapshots, which can be represented as $TEG = \{S_i\}$ where i is an increasing integer from 0. Each snapshot $S_i = (V_i, E_i)$ represents the state of the graph at a specific time point during the evolution process, where V_i is the vertex set of S_i, E_i is the directed edge set of S_i. For example, the time-evolving graph in Fig. 1 has three snapshots. Time-evolving graphs can be used to predict future trends in the real world, such as the trend of the epidemic and pandemic, by mining the inherent law among the historic snapshots of the real-world graph [3]. A TEG analysis algorithm typically works in two steps: (1) executing a same static graph algorithm on each snapshot separately; (2) using a certain machine learning algorithm, based on the computation results of the snapshots, to study the development law of real world and predict the future trend.

In order to support the *TEG* analysis algorithms efficiently, an efficient time-evolving graph storage system is very important, which should be well designed to collect and store the large

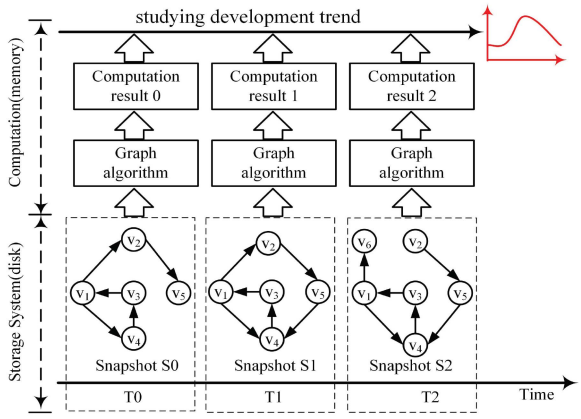


Fig. 1. Time-evolving graph with three snapshots.

number of snapshots generated continuously as time evolves, in a way similar to the case of relational database (DBMS). *Storage cost* and *snapshot access speed* are the two most important and usually conflicting indicators when designing a time-evolving graph storage system [4]. In Sections II-B and II-C, we will discuss the previous works that are most relevant to TgStore.

B. Single-Snapshot Oriented Methods

Google published its graph analysis system Pregel [1] in 2010. Since then, there has been a surge of interest in graph analysis systems in both academia and industry. Most of them, such as Pregel [1], GraphChi [15] and BlizG [14], aim to speed up the analysis task of static graph, by improving the efficiencies of computation, Disk I/O and communication. Usually, we call these systems *static graph analysis systems*.

A static graph is essentially a snapshot of a time-evolving graph at a certain time point. There are three key limitations when a static graph analysis system executes a time-evolving graph analysis task. First, each snapshot is stored independently by using a certain data structure, typically, the edgelist [4]. However, in order to support the time-evolving graph analysis task, a large number of snapshots should be collected continuously from the external business systems, resulting in a very high space cost. Second, when executing a time-evolving graph analysis task, each snapshot needs to be loaded into memory independently, leading to a long loading time for disk I/Os. Third, the high per-snapshot storage cost limits the number of snapshots that can be loaded into memory simultaneously, reducing the performance of some efficient computation methods [3].

C. Multiple-Snapshots Oriented Methods

Storing a time-ordered sequence of snapshots naively, e.g., as in Yang's model [16], and Rossi's model [17], would clearly require a prohibitively large storage capability. A few recent papers address the issues of storage and retrieval in time-evolving graphs. DeltaGraph [18] designs an index data structure that compactly stores the history of all changes in a dynamic graph and provides efficient snapshot reconstruction. It represents each snapshot by storing the incremental data difference (Delta) between the baseline and the current snapshots, aiming to reduce

the average snapshot storage cost. Variants of this strategy have been presented in Koloniari's model [9], [10], Khurana's model [11] and Chronos' model [3]. However, there are still a large number of vertices and edges that are stored repeatedly in Deltas, since vertices and edges can be deleted and then recovered in any subsequent snapshots. G*[19] compresses dynamic graph data based on commonalities among the graphs in the series for deduplicated storage on multiple servers. However, it was designed mainly to support complex queries on large graphs. LLAMA [12] uses multiversioned arrays to represent a mutating graph, but their focus is primarily on in-memory representation. There is also recent work on streaming analytics over dynamic graph data [20], [21], but it typically focuses on analyzing only the recent activities in the network.

Another challenge to designing a time-evolving graph storage system is the data structure that organizes the data for storage and retrieval, since it should support the appending of new snapshots efficiently. APMA [8] introduces *an adaptive packed-memory array* to deploy time-evolving graphs. It handles this problem by reserving storage space for the vertices and edges of the new snapshot. This comes at the storage cost for the reserved storage space. The efficiency of new snapshot appending is also the problem that the recent work Graphone is facing when the edge log data of the new snapshot is moved to the adjacency store to create a new adjacency list version [4]. This process is time-consuming [8].

D. Motivation

In summary, for large time-evolving graphs, most existing graph systems store and process each snapshot separately, resulting in costly storage overhead of average snapshot and long time for loading and computing. They are inefficient in terms of storage capacity. Several storage-oriented graph systems have been proposed to address the problems, by using different methods. However, key challenges, such as average snapshot storage cost and snapshot access speed, remain inadequately addressed.

The discussion and analysis above motivate us to present TgStore that can effectively support analysis tasks for large time-evolving graphs by efficiently storing a large number of snapshots obtained continuously from external business systems.

III. System Design and Implementation

We first introduce the snapshot-level data deduplication of TgStore in Section III-A. The efficient representation of structure-changing of the snapshot-level data deduplication in Section III-B. We further present the optimizations in Section III-D, and present several functions designed to facilitate the design of time-evolving graph algorithms over TgStore in Section III-E and a summary in Section III-F.

A. Snapshot-Level Data Deduplication

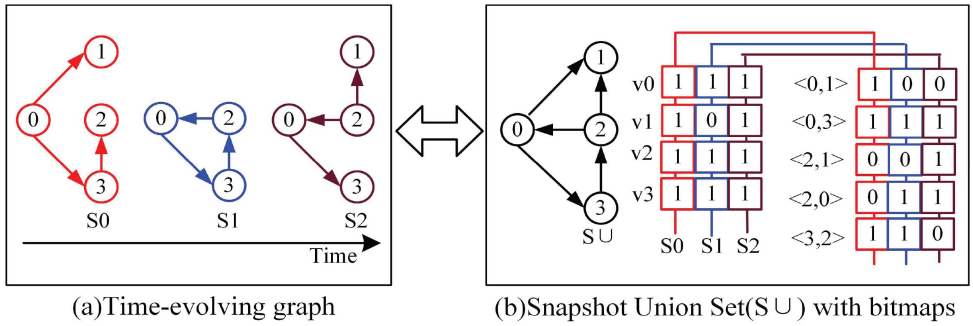
Investigation of Time-evolving Graphs: Real-world time-evolving graphs have an important feature of snapshot similarity that has been reported in several recent studies [22]. However, most of their results have been derived from the time-evolving graph datasets that were generated artificially based on the static graphs [4], [12]. This method does not help us study the features of real-world time-evolving graphs, further hindering the design and implementation of efficient time-evolving graph storage systems.

To address this problem, we collect three types of datasets, that is, the social network Weibo dataset, the web U.K. dataset[2,3] and the collaboration network Coauthor dataset.[4] Weibo has 27 snapshots with an average 4.18×10^8 edges per snapshot; U.K. has 31 snapshots with an average 3.72×10^9 edges per snapshot; Coauthor has 27 snapshots with an average 2.12×10^6 edges per snapshot. The time interval between any two consecutive snapshots is one day, one day and one year respectively for Weibo, U.K. and Coauthor. By analyzing the datasets, we find that the difference between any two consecutive snapshots is small. For the case of Weibo, the difference any two consecutive snapshots is only about 0.09%. This leads to a high ratio of repeated vertices and edges among the snapshots of a time-evolving graph. Since the number of publicly available time-evolving graph datasets is extremely small, some related works, such as LLAMA [12], resorted to generating snapshots based on static graphs. Like LLAMA, we also generate a time-evolving graph dataset with 100 snapshots based on the static graph Twitter.[5] The difference between any two consecutive snapshots is less than 0.22%, which is larger than the case of Weibo dataset.

Snapshot Union Set: Log-based method is effective to avoid the repeated vertices and edges between any two consecutive snapshots [9], [10]. Since the *log* can be easily obtained from the external business system by recording the deleted and added vertices and edges relative to the last snapshot. However, by analyzing the datasets, we find that there are still a large number of repeated vertices and edges in the *log* since vertices and edges can be removed from one snapshot and added back soon in one of the subsequent snapshots. Furthermore, it requires a long time to reconstruct a snapshot by using the baseline snapshot and the *log* between them.

Different from Log-based methods, we employ snapshot union set (S∪) [5] to represent a time-evolving graph. As shown in Fig. 2, the snapshot union set (S∪) can be obtained by executing the union operation on all the snapshots of the time-evolving graph. The union operation eliminates all the repeated vertices and edges among the snapshots, resulting in a very low storage overhead of an average snapshot. As shown in Table I, the snapshot union set (S∪) of Weibo is only 1.012× larger than the average snapshot of the original time-evolving graph. This method is inspired by W. Huo and V. J. Tsotras [5] who use snapshot union set to compute the lower limit of the shortest path value of the snapshots. Different from them, we employ snapshot union set to reduce the storage cost of an average snapshot, aiming to design an efficient time-evolving graph storage system. In Table I, the expansion of edges is defined as

[2][Online]. Available: https://law.di.unimi.it/webdata/ UK-2007-03/
[3][Online]. Available: https://law.di.unimi.it/webdata/UK-2007-04/
[4][Online]. Available: https://www.aminer.cn/
[5][Online]. Available: http://konect.cc/networks/twitter/



(a)Time-evolving graph (b)Snapshot Union Set(S∪) with bitmaps

Fig. 2. Snapshot union set ($S\cup$) with bitmaps.

TABLE I
INVESTIGATION OF TIME-EVOLVING GRAPHS

DataSet	Weibo	UK	Coauthor	Twitter
Type	Social Net.	Web	Collab. net.	Social Net.
# of snapshots	27	31	27	100
Interval	1 day	1 day	1 year	-
Edges/snapshot	$\mathbf{4.18\times10^8}$	$\mathbf{3.72\times10^9}$	$\mathbf{2.12\times10^6}$	$\mathbf{1.32\times10^9}$
Edges (total)	1.129×10^{10}	1.153×10^{11}	5.724×10^7	1.334×10^{11}
Vertices/snapshot	$\mathbf{1.78\times10^6}$	$\mathbf{1.07\times10^8}$	$\mathbf{1.62\times10^6}$	$\mathbf{4.16\times10^7}$
Vertices (total)	4.80×10^7	3.31×10^9	4.37×10^7	4.46×10^9
Edges (union set)	$\mathbf{4.23\times10^8}$	$\mathbf{7.36\times10^9}$	$\mathbf{6.72\times10^6}$	$\mathbf{1.47\times10^9}$
Expansion (times)	**1.012**×	**1.980**×	**3.170**×	**1.111**×
Vertices (union set)	$\mathbf{1.78\times10^6}$	$\mathbf{1.07\times10^8}$	$\mathbf{1.63\times10^6}$	$\mathbf{4.16\times10^7}$
Expansion (times)	**1.000**×	**1.000**×	**1.006**×	**1.000**×

the ratio of the number of edges of the snapshot union set to the number edges of an average snapshot. Similarly, the expansion of vertices is defined as the ratio of the number of vertices of the snapshot union set to the number vertices of an average snapshot.

Snapshots Reconstructing: A time-evolving graph storage system should support the application algorithms to obtain any snapshots from the system rapidly. In order to achieve this goal, we design a bitmap for each vertex or edge of the snapshot union set (S∪), as shown in Fig. 2. Each bit of the bitmap for a vertex/edge corresponds to a snapshot and indicates the presence/absence (1/0) of this vertex/edge in the snapshot, making it very efficient (with a simple operation) to determine if a vertex or edge belongs to a certain snapshot or not. Furthermore, as shown in Table I, the snapshot union set (S∪) is only slightly larger than an average snapshot. As a result, facilitated by the snapshot union set (S∪), very few irrelevant vertices and edges will be accessed when reconstructing multiple snapshots to load into memory, a requirement by most time-evolving graph algorithms, leading a high speed of snapshot accesses.

Challenges: By using the snapshot union set (S∪) with the bitmaps, we can store time-evolving graphs with a very low storage overhead and reconstruct snapshots at a high speed. However, the graph structure of the snapshot union set (S∪) is dynamically changing when it is appended with new snapshots that come from the external business system. In order to append a new snapshot, a typical data structure for the snapshot union set (S∪) representing a dynamically changing graph usually needs to move or reconstruct data, resulting in a long appending time [8]. Let us assume that we use the data structure of *edgelist* [4] to store the snapshot union set (S∪). When the new edges from the new snapshot arrive, a number of old edges in the *edgelist* file need to be moved to accommodate the new edges in such a way that the edges of the *edgelist* file are kept in order, reducing the speed of the snapshot appending. The ordered edges make it easy for each deleted edge to locate its exact position in its bitmap where a bit of the value of 0 will be appended.

B. Efficient Representation of Structure-Changing S∪

In order to address the challenges identified above, we propose an efficient representation for the snapshot union set (S∪), as shown in Fig. 3.

Design of Data Structure: A new snapshot coming from the external business system is captured in TgStore by using a *logfile* that records the deleted and added vertices and edges from the last snapshot. The reason we adopt this method is that recording the difference between the last snapshot and the new snapshot is easily done by the external business system [8].

A *Fundamental Edge Data Block (FEDB)* is created first for a time-evolving graph when executing the first operation of appending a snapshot. Different from subsequent operations of snapshot appending, we need to store all the edges of the initial snapshot (S_0) into the *FEDB*. Each edge is stored together

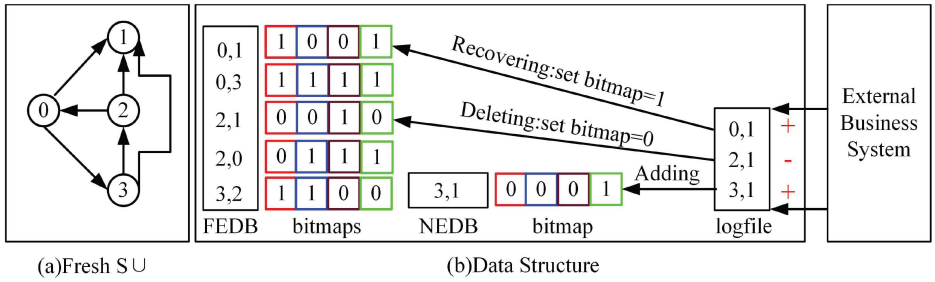


Fig. 3. Representation of structure-changing S∪.

with its bitmap. We do not store the vertices since they are implied by the information on edges. When the time-evolving graph analysis algorithms load the edges of S∪ into memory, information on vertices can be recovered to construct any data structure in the memory according to the algorithms.

A *New Edge Data Block (NEDB)* is created when a new snapshot arrives with the new *logfile*. It updates the snapshot union set, together with the bitmaps of *FEDB* and its previous *NEDB*s. The *NEDB* only stores the edges that do not exist in the old snapshot union set since each deleted or recovered edge has existed in the *FEDB* or one of the previous *NEDB*s. For each deleted edge, we only need to locate the edge, and then append to the corresponding bitmap a bit with the value of 0. For the edge that has existed in the old snapshot union set and is recovered in the new snapshot, we only need to locate the edge, and then append a bit with the value of 1 to the bitmap. Our method is different from the log-based method [8] and DeltaGraph [18] that need to store the difference between the new snapshot and the last snapshot. For each of other edges that do not change, we only need to append a bit to its bitmap, with the same value of the last bit.

By using the above design, we do not need to move any edges when a new snapshot is appended.

C. Snapshot Appending

As a time-evolving graph storage system, the most important role of TgStore is to store states (snapshots) of the graph at different time points, and provide the states to time-evolving graph algorithms. With the time evolving, the time-evolving graph is updated by appending new snapshots. The efficiency is important for TgStore. We discuss the snapshot appending in this Section.

In-Memory Snapshot Appending: TgStore represents a time-evolving graph with N snapshots by using one *FEDB* and N-1 *NEDB*s. They are stored in external storage, typically, the disk. When appending a new snapshot, the new *logfile* includes two types of operation, that is, deleting and adding an edge. For each deleted edge, we need to locate the edge in the *FEDB* or one of the *NEDB*s, and then set the new bit of the bitmap of this edge as 0. This introduces one random access to search the first position of the edge group of source vertex of the deleted edge, and then a number of random accesses to search the exact position in the edge group. For each added edge, we need to check whether the edge has existed in the old snapshot union set (S∪). Random accesses also occur in the case of deleted edge. This slows down the snapshot appending if the number of added and deleted edges is sufficiently large.

To address this problem, we execute the task of snapshot appending in memory. When a new *logfile* arrives, we load the *FEDB*, *NEDB*s and the *logfile* into the memory together. We separate the edge bitmaps from the *FEDB* and *NEDB*s by using edge bitmap files that are also loaded into the memory. This design improves the snapshot appending speed significantly. Since we only need to load *FEDB* and *NEDB*s from disk to build the indexes that help find the bitmap position of an added or deleted edge. The loaded *FEDB* and *NEDB*s need not to be written back to disk since they are not changed during the snapshot appending. We only need to write the bitmap blocks and the new *NEDB* back to the disk. This process is efficient since the bitmap blocks and the new *NEDB* are small. More importantly, they are written back sequentially.

Self-Adaptive Snapshot Appending: The design of in-memory snapshot appending can speed up the process of snapshot appending significantly. However, the problem of random accesses still exists in the memory. When the number of added and deleted edges in the *logfile* is large, the efficiency of snapshot appending is still limited. Since random accesses in memory is 10× slower than their sequential counterparts [23].

Hence, we further optimize TgStore for the case of large numbers of deleted and added edges as follows. Each block of *FEDB*, *NEDB*s and *logfile* has a pointer that points to the first position respectively. The edges in each block are ordered according to the source vertex and then the destination vertex. This can be easily done by keeping the edges of each *logfile* in order. We compare the current edge of the *logfile* block with other edges pointed to by other pointers; if we find one edge that is "equal" to the current edge of *logfile* block, only one operation is needed to set the bit value to 0 or 1, depending on whether the edge is deleted or added, and then the pointer of the *logfile* block moves to next position; if the current edge of *logfile* block is "less than" any of the edges pointed to by other pointers, again only one operation is needed to append this edge to the new *NEDB*, indicating that this edge does not exist in the old snapshot union; otherwise, the pointers of *FEDB* and *NEDB*s move to the next position. This process is repeated until each of the added or deleted edge in the *logfile* block has been processed.

This method needs to access the entire blocks of *FEDB*, *NEDB*s and *logfile*. TgStore first evaluates the time that this

process takes. It also evaluates the time for processing one edge in the *logfile* by using the random access mode. By using the two evaluation values, we can calculate a threshold θ. When the number of added and deleted edges in the *logfile* block is less than θ, TgStore adopts the random access mode, otherwise the batched access mode.

Single Updating: As discussed before, the time-evolving graph is batch updated when appending a new snapshot. The appended snapshots represent the historic states of the graph. Hence, the case rarely happens to update the historic snapshots, including the graph structure of the snapshot and the values of the vertices and edges. however, in some cases, updating is also needed for historic snapshots. For example, wrong value of one vertex was introduced before the snapshot appending. Hence, TgStore can also update the state of one object (vertex/edge) in one snapshot, by moving or adding the object and updating the value of the object. Specifically, when users need to update one object, TgStore only needs to locate the object and then updates the bit of the bitmap for the object according to the adding operation and the deleting operation, or updates the value of the object directly.

Traditional relational database management systems (DBMS) usually adopt the checkpoint idea which guarantees that all the operations (insert, update and delete) between any two consecutive checkpoints can be written back to the disk successfully. Each checkpoint represents a state of the relational database management system. The process of snapshot appending in TgStore is similar to the checkpoint idea. TgStore appends a new snapshot according to a *logfile* that records the deleted and added vertices and edges from the last snapshot. Each snapshot represents a state of the time-evolving graph. In order to save storage space, a *logfile* can be deleted after being used to create a new snapshot.

D. Optimizations

We further optimize TgStore as follows.

Design of Edge Data Structure: We initially adopt the *edgelist* [4] structure to store edge data. The advantage of the *edgelist* structure is that each edge data with source vertex and destination vertex has the same size, making it easy to locate each edge. As mentioned before, the edges in each block of *FEDB*, *NEDB*s and *logfile* are sorted by the source vertex and then destination vertex. However, the source vertex is stored repeatedly in each edge group. To address this problem, we remove the source vertices from the data blocks, by using a pointer for each group to identify the first position of the edge group. This method is simple but effective, which further reduces the storage cost of an average snapshot, and similar to the compressed sparse row (CSR) or linked list [4].

Balancing Storage Cost and Snapshot Access Speed: For most time-evolving graphs, such as social networks, the size of the snapshot union (S$\cup$) increases slowly when the business reaches a steady state. However, for some other time-evolving graphs, such as web graph, the size of the snapshot union (S$\cup$) increases continuously at a certain rate because new webpages and links are generated continuously at a steady pace. In this case, the snapshot access will slow down with time, including the snapshot appending and reconstructing.

To address this problem, TgStore allows users to split the snapshots of a time-evolving graph into snapshot segments, according to the features of business. Each segment with consecutive snapshots is stored by using the storage scheme discussed above. The last segment is responsible for the newly appended snapshot until a new segment is generated.

E. Facilitating Functions

TgStore is a time-evolving graph storage system that has been designed and implemented to efficiently support time-evolving graph algorithms. We present several functions designed to facilitate the design of time-evolving graph algorithms over TgStore.

Snapshots Loading: This API helps time-evolving graph algorithms load a snapshot subset of a time-evolving graph into memory. The bitmap files of *FEDB* and *NEDB*s are first loaded into memory. To improve the memory usage, irrelevant bits in each bitmap are removed. Then, a vertex list with N elements is created in memory, where N is the number of vertices of the snapshot union (S$\cup$) of the loaded snapshots. This helps algorithms locate each vertex easily, according to the vertex ID. Finally, this API begins to load each relevant edge whose bitmap contains at least one bit with value of 1. During this process, each vertex obtains an address that points to its first edge. This API also evaluates the needed memory space of the snapshot subset. If the memory capacity is insufficient, the snapshot subset will be divided into smaller subsets. In this case, the execution task of the time-evolving graph algorithm will also be divided into tasks corresponding to the number of the smaller subsets.

At this point, the memory data of the needed snapshots is ready for the time-evolving graph algorithm.

Snapshot-Level Incremental Computing: Like several time-evolving graph analysis systems, such as Chronos [3], TgStore also supports snapshot-level incremental computing to obtain a higher execution speed of the time-evolving graph algorithm that regularly needs to execute the same static graph algorithm on different snapshots. In this case, the baseline snapshot, typically, the first snapshot S_0 or the snapshot intersection (S$\cap$) [3], is selected first, and then executed by using a certain static graph algorithm. Other snapshots can be executed based on the computation result of the baseline snapshot to speed up the algorithm convergence.

Computation Models: Like many graph systems, TgStore adopts the bulk synchronous parallel (BSP) computation model [1], that is, all the vertices are executed in parallel. When executing each vertex, there are two alternative modes, i.e., *push* and *pull* [24]. The push mode and pull mode are suitable for different graph algorithms. The pull mode is efficient for the graph algorithms, such as Pagerank [25], where most vertices need to interact with its neighbors. The push mode is efficient for the graph algorithms, such as Single Source Shortest Paths (SSSP) [26], where only a small number of vertices need to interact with its neighbors.

Algorithm Implementations: We implemented several time-evolving graph algorithms, including SSSP [26] and Pagerank [25], to evaluate Tgstore. The SSSP algorithm computes the distance of the shortest path from a given source vertex u to each other vertex in each snapshot. The PageRank algorithm is used to rank websites in each snapshot. Algorithm 1 illustrates the execution process of SSSP. The input of the time-evolving graph algorithm of SSSP is a snapshot set $TEG = \{S_0, S_1, \ldots, S_{N-1}\}$. Each snapshot can be defined as $S_i = (V_i, E_i)$, where $0 \leq i \leq$ N, V_i is the vertex set of S_i, E_i is the directed edge set of S_i. After the snapshot set has been loaded into memory, the first task is to execute the baseline snapshot (the snapshot intersection $S\cap$) (Lines 2–18). In the initialization stage (Lines 2–7), the value of source vertex vs is set to 0, the values of other vertices that belong to the baseline snapshot are set to $+\infty$; this means that the distance from the source vertex vs to any one of other vertices is currently unknown. In the computation stage, the computation task proceeds with a number of supersteps (Lines 8∼18); in each superstep, all vertices (excluding vs) that belong to the baseline snapshot are executed in parallel (Lines 9∼16); when executing each vertex v (Lines 9∼17), there is one *push* operation for each outgoing neighbor v'; the *push* operation updates the value of v' if a shorter path has been received; the updated values of v' will be used in next superstep by its outgoing neighbors. The graph computation job ends at the superstep where each vertex does not receive a shorter distance. After the execution of baseline snapshot finished, each of other snapshots is initialized by using the computation results of the baseline snapshot (Lines 19–34). Then, each snapshot with the computation results of baseline snapshot is executed to obtain the single source paths (Lines 35–50).

Algorithm 1: Time-Evolving Graph Algorithm of SSSP.

```
   Input: N; TEG = {S_i | 0 ≤ i < N};
   Output: sssps of each snapshot S_i
   /* S_i is ith snapshot of TEG                    */
1  Loading(S_0, S_1, ..., S_{N-1});
   /* Run sssp on snapshot intersection S∩
      (baseline snapshot)                           */
   /* S∪ is the snapshot union of TEG               */
2  for each vertex v ∈ S∪ do
3      if each v.bitmap[i] == 1 , 0 ≤ i < N then
4          v.u[b] = +∞;
5      end
6  end
7  v0.u[b] = 0;
8  while S∩ has active vertices do
9      for each vertex v ∈ S∩ and v.active = 1 do
10         v.active = 0;
11         for each outgoing vertex v' of v do
12             if v'.u[b] > v.u[b] + e.u[b] then
13                 v'.u[b] = v.u[b] + e.u[b];
14                 v'.active = 1;
15             end
16         end
17     end
18 end
   /* Run sssp on Snapshot(S_i) based on the
      computation result of baseline                */
19 for each vertex v ∈ S∪ do
20     for i = 0; i < N; i++ do
21         if v.bitmap[i] == 0 then
22             Continue;
23         end
24         if v ∈ S∩ then
25             v.u[i] = v.u[b];
26         end
27         else
28             v.u[i] = +∞;
29         end
30     end
31 end
32 for i = 0; i < N; i++ do
33     v0.u[i] = 0;
34 end
35 for i = 0; i < N; i++ do
36     while S_i has active vertices do
37         convergenceFlag = 1;
38         for each vertex v ∈ S∪ and v.active = 1 do
39             v.active = 0;
40             if v.bitmap[i] == 1 then
41                 for each outgoing vertex v' of v do
42                     if v'.bitmap[i] == 1 and v'.u[i] >
                       v.u[i] + e.u[i] then
43                         v'.u[i] = v.u[i] + e.u[i];
44                         v'.active = 1;
45                     end
46                 end
47             end
48         end
49     end
50 end
```

F. Summary

To sum up, TgStore is an efficient time-evolving graph storage system. We first design the Snapshot-level Data Deduplication (SLDD) to reduce the per-snapshot storage cost, by using a union of all snapshots with one bitmap for each object (vertex or edge) to avoid the repeated objects among snapshots of the time-evolving graph. We then design the Structure-Changing Graph Representation (SCGR) to address the data movement problem of SLDD, by using two key data structures of *Fundamental Edge Data Block (FEDB)* and *New Edge Data Block (NEDB)*. This design results in high-speed snapshot appending. To the best of our knowledge, there is no existing work that uses these techniques to design and implement a time-evolving graph storage system.

IV. Experimental Evaluation

A. Experimental Setup

In this section, we conduct extensive experiments to evaluate the performance of TgStore. Experiments are conducted on the Intel machine that has 48 E5-2650 cores with 256 GB of RAM. We study TgStore in terms of storage costs, snapshot access speed and the execution efficiency of algorithms.

*Datasets:*We use four time-evolving graph datasets that are summarized in Table I. Weibo is a social network, which has 27 snapshots with an average 1.78×10^6 vertices and 4.18×10^8 edges per snapshot; The time interval between two consecutive snapshots of Weibo is one day. U.K. is a Web dataset, which has 31 snapshots with an average 1.07×10^8 vertices and 3.72×10^9 edges per snapshot; The time interval between

two consecutive snapshots of U.K. is one day. Coauthor is a collaborative network, which has 27 snapshots with an average 1.62×10^6 vertices and 2.12×10^6 edges per snapshot; The time interval between two consecutive snapshots of Coauthor is one year. As mentioned in Section III, since the number of publicly available time-evolving graph datasets is extremely small, some related works, such as LLAMA [12], resorted to generating snapshots based on static graphs. Like LLAMA, we also generate a time-evolving graph dataset with 100 snapshots based on the static graph Twitter [27]. The difference between any two consecutive snapshots is less than 0.22%, which is larger than the case of Weibo dataset (0.09%).

Algorithms and Baselines: We implement three typical time-evolving graph algorithms that are used frequently by recent studies, PageRank (PR) [25], Single Source Shortest Paths (SSSP) [26] and Breadth First Search (BFS) [28], to evaluate algorithm execution efficiency of TgStore. The PageRank algorithm is used by Google Search to rank websites in their search engine. The SSSP algorithm computes the distance of the shortest path from a given source vertex u to each other vertex in a graph. The BFS algorithm is a building block of more advanced algorithms that involve graph traversals, such as betweenness centrality and community detection. In this part, we compare TgStore with two state-of-the-art in-memory time-evolving graph processing systems, Graphone [4] and LLAMA [12].

B. Storage Costs

Ratio of Repeated Edges: We first study the ratio of repeated edges of real-world time-evolving graph datasets. Vertices are not discussed since their numbers are small in all the three datasets, compared with their edge counterparts. By analyzing the three real-world time-evolving graph datasets, we find that any two consecutive snapshots have a high similarity. We define the similarity of two snapshots, S_i and S_j, as follows.

$$Sim(S_i, S_j) = \frac{Ne(S_i \cap S_j) \times 2}{Ne(S_i) + Ne(S_j)}$$

where $Ne(S_i)$ and $Ne(S_j)$ are the numbers of edges of snapshots S_i and S_j respectively. $Ne(S_i \cap S_j)$ is the number of the common edges of S_i and S_j. Experimental results show that real-world time-evolving graphs have high snapshot similarity. For the case of Weibo, the snapshot similarity of any two consecutive snapshots, $Sim(S_i, S_{i+1})$, is over 99%, with an average of 99.9%. The average snapshot similarity of U.K. is over 96.5%, which is slightly lower than Weibo. The reason is that, for the web graph U.K., web pages and links among them are generated continuously at a steady pace. The snapshot similarity for Coauthor is 90.8%, and lower than both Weibo and U.K., because the time interval of any two consecutive snapshots in the former is one year, much longer than that in the latter. The high snapshot similarity indicates that there are a large number of repeated edges among the snapshots of the real-world time-evolving graphs.

Effectiveness of Snapshot-Level Data Deduplication: We conduct experiments to study the effectiveness of *Snapshot-level Data Deduplication* by using the collected datasets of real-world time-evolving graphs. For each dataset, we record the number of edges of the snapshot union set (S∪) with the number of snapshots ranging from 1 to the number of the snapshots of the dataset, that is, 27, 31, 27 and 100 respectively for Weibo, U.K., Coauthor and Twitter. We take the number of edges of average snapshot as baseline storage cost, without the *Snapshot-level Data Deduplication*. By using the *Snapshot-level Data Deduplication*, we divide the number of edges of the snapshot union set (S∪) by the number of snapshots, together with the additional storage cost of bitmaps, to obtain the average snapshot storage cost.

Experimental results indicate that, for the case of Weibo, the size of the snapshot union (S∪) is increasing slowly with the number of snapshots, with a trend of convergence. The size of S∪ is only 1.012× larger than the size of the average snapshot of the original time-evolving graph when the number of snapshots is 27, resulting the compression ratio of 26.68:1. There is a slight difference for the U.K.. In this case, the size of the snapshot union set (S∪) is increasing slowly with the number of snapshots with a ratio. When the number of snapshots reaches 31, the size of the snapshot union set (S∪) is also only 1.980× larger than the size of the average snapshot of the original time-evolving graph, resulting in a compression ratio of 15.65:1. As mentioned in Section III, we initially store the edges of the snapshot union set (S∪) of a time-evolving graph by using the *edgelist* data structure that is usually employed by many graph analysis systems [15]. When computing the storage cost of each original snapshot to obtain the compression ratio, *edgelist* data structure is used for fair comparison. In fact, the compression ratio is dominated mainly by the number of the repeated vertices and edges among the snapshots. The reason is that, for the same representation of graph, the storage cost of an average edge is also the same. Similarly, as shown in Table II, the compression ratio of Coauthor is 8.52:1. Compared with the three datasets above, the time-evolving graph Twitter has a higher compression ratio of 37.42:1. The reason is that it has a larger number of snapshots with a slightly higher similarity.

Effectiveness of Structure-Changing Graph Representation: As mentioned in Section III, by using the *edgelist* data structure, the source vertex of the snapshot union set (S∪) is stored repeatedly in each edge group. In order to address this problem, we remove the source vertices from the data blocks, by using a pointer for each group to identify the first position of the edge group. We also conduct experiments to evaluate the effectiveness of this optimization.

This method is similar to the compressed sparse row (CSR) or linked list [4], which can theoretically compress the snapshot union set (S∪) by a ratio of near 2:1. However, experimental results show that it obtains a compression ratio of 1.48:1 on the dataset of Weibo. Since the gain of the reduced storage space depends on the number of the out-edge neighbors of each source vertex. If a source vertex has no out-edge neighbors, the gain is negative since there is a fixed cost for each edge group to store the first position. The compression ratio is 1.19:1 in case of the dataset U.K.. The reason is that, compared with Weibo, the snapshot union set (S∪) of U.K. has a larger number edges, requiring a larger space (8 bytes) to store the pointer for each

TABLE II
COMPRESSION RATIO & AND AVERAGE EDGE COST

DataSet	Weibo	UK	Coauthor	Twitter
Comp. ratio (union set)	26.68:1	15.65:1	8.52:1	37.42:1
Comp. ratio (opt.)	1.48:1	1.19:1	1:1	1.15:1
Comp. ratio (total)	**39.49:1**	**18.62:1**	**8.52:1**	**43.03:1**
Av. edge cost	**2.81 bits**	**5.21 bits**	**8.69 bits**	**1.48 bits**

edge group. The compression ratio of Coauthor is 1:1. Since the gain is negative in this case, TgStore abandons this optimization. Experimental results also show that Twitter has a compression of 1.15:1.

Total Compression Ratio: By combining *Snapshot-level Data Deduplication* and the optimization of edge data structure, Tgstore obtains a high compression ratio. As shown in Table II, the compression ratio of Twitter is 43.03:1, with an average edge storage cost of 1.48 bits. Graphone and LLAMA are excluded in these experiments, since they store the snapshots of the time-evolving graph without compression. We do not think it is appropriate to compare them with TgStore in terms of storage cost. Hence, we use the total compression ratio to study the storage cost of TgStore here.

C. Snapshot Access Speed

We study the snapshot access speed of TgStore in terms of snapshot appending and loading.

Snapshot Appending: Experiments are conducted on the datasets of Weibo, U.K., Coauthor and Twitter. We first created the snapshot union set (S∪) and the bitmaps of each dataset using the first snapshot. New snapshots then are appended to the snapshot union set (S∪) one by one. The number of new snapshots is 26, 30, 26 and 99, respectively for the datasets of Weibo, U.K., Coauthor and Twitter. In order to study the effectiveness of the self-adaptive appending, each experiment is repeated by using the random access mode and batched access mode. Experimental results show that, for the Weibo, the average snapshot appending time of the random access mode is 20.38 seconds, and the time is 20.49 seconds for the batched access mode. The experimental results indicate that the random access mode is slightly faster than the batched access mode. For the Twitter, the batched access mode is 1.06× faster than the random access mode. However, for the U.K., the average snapshot appending time of the random access mode is 437.17 seconds, and the time is 348.05 seconds for the batched access mode. In this case, the batched access mode is 1.26× faster than the random access mode. The reason is that, compared with Weibo, the *log* of each new snapshot has a larger number of added and deleted edges, enabling the batched access mode to work well. As shown in Fig. 4, for the Coauthor, the batched access mode is 1.56× faster than the random access mode. The experimental results indicate that the batched access mode is effective.

When we append a new snapshot to the snapshot union set (S∪), the time of an average update (adding or deleting an edge) is only 1.8*us* for the random access mode, and 1.43*us* for the batched access mode. The high speed of TgStore stems from

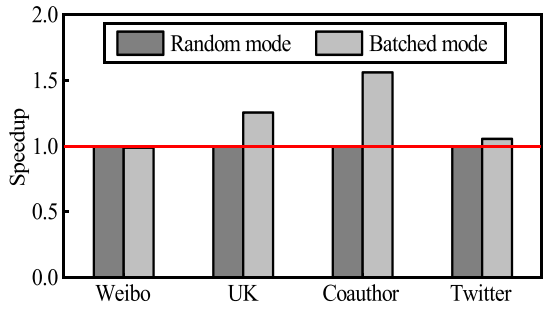


Fig. 4. Speed of snapshot appending.

two parts, i.e., the *Structure-Changing Graph Representation* and the batched access mode.

Snapshots Loading: For a graph analysis task, the total execution time is dominated heavily by the loading time [15]. Hence, we conducted experiments to study TgStore in terms of the speed of snapshots loading, by using the time-evolving graphs of Weibo and Twitter that have been stored in TgStore.

Experiment is conducted repeatedly by loading the snapshots from the time-evolving graph of Weibo, with the number of snapshots varying from 1 to 27. We compare TgStore with LLAMA [12] and Graphone [4], two state-of-the-art graph storage systems. When loading the first snapshot of Weibo, the loading time is 276.77 seconds, 78.15 seconds and 11.00 seconds for Graphone, LLAMA and TgStore respectively. Tgstore is 25.2× and 7.1× faster than Graphone and LLAMA respectively. It also obtains a significant performance improvement over Graphone and LLAMA. As shown Fig. 5(a), at the point of 27 snapshots, TgStore is 5.9× and 21.3× faster than LLAMA and Graphone respectively. Experiment is also conducted repeatedly by loading the snapshots from the time-evolving graph of Twitter, with the number of snapshots varying from 1 to 100. As shown in Fig. 5(d), TgStore also obtains a significant performance improvement over Graphone and LLAMA. For example, at the point of 100 snapshots, Tgstore is 16.5× and 6.3× faster than Graphone and LLAMA respectively.

We further conducted experiment repeatedly by loading the snapshots from the time-evolving graph of U.K., with the number of snapshots varying from 1 to 31. As shown in Fig. 5(b), TgStore also obtains a significant performance improvement over Graphone and LLAMA. Experiment is also conducted repeatedly by loading the snapshots from the time-evolving graph of Coauthor, with the number of snapshots varying from 1 to 27. As shown in Fig. 5(c), at the point of one snapshot, the loading time is 0.06 seconds, 0.56 seconds and 0.17 seconds for Graphone, LLAMA and TgStore respectively. TgStore is

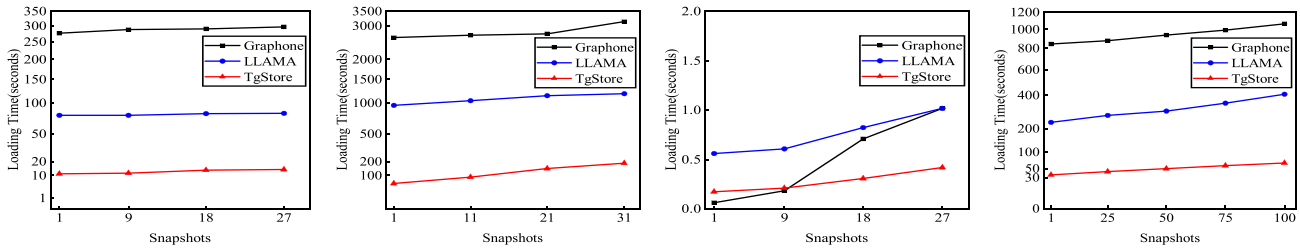


Fig. 5. Loading efficiency.

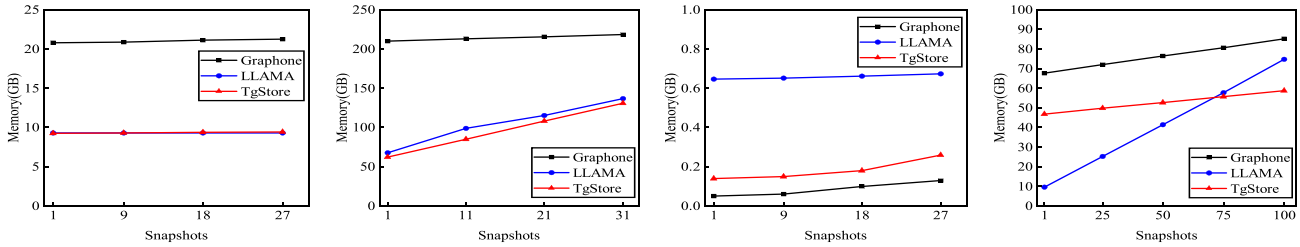


Fig. 6. Memory usage.

3.29× faster than LLAMA, and 2.83× slower than Graphone. However, the loading efficiency of Graphone is reduced significantly with the number of snapshots varying from 9 to 27. Different from Graphone, TgStore can obtain a high performance when loading more snapshots. At the point of 27 snapshots, Tgstore is 2.38× faster than Graphone and LLAMA. The experimental results show that our time-evolving graph storage scheme is effective, which can provide a high speed of snapshots loading.

D. Execution Efficiency of Algorithms

We study the algorithm execution efficiency that is provided by TgStore, in terms of memory usage, algorithm execution speed and the effectiveness of incremental computing, by comparing with two state-of-the-art time-evolving graph storage systems, LLAMA [12] and Graphone [4].

Memory Usage: In each experiment of LLAMA, Graphone and TgStore, we load different numbers of Twitter snapshots into memory to study the memory usage. Experimental results show that when the number of loaded snapshots is less than 75, LLAMA outperforms TgStore in terms of memory usage. The reason is that, TgStore is designed to store time-evolving graphs with a large number of snapshots, and support the fast reconstructing of any snapshot in memory. When the number of snapshots loaded into the memory is small, the gain of the bitmap-based designs can be negative. However, the memory usage of TgStore is significantly higher than that of Graphone when loading snapshots with different number. As shown in Fig.–6(d), TgStore outperforms LLAMA and Graphone by 1.27× and 1.45× respectively in terms of memory usage when the number of loaded snapshots is 100. The high memory usage enables TgStore to load a larger number of snapshots into memory simultaneously, leading to higher execution speed that come from the snapshot-level incremental computation and higher memory access locality [3].

We also conducted experiments by loading different numbers of Weibo snapshots into memory to study the memory usage. As show in Fig. 6(a), the memory usage of TgStore is similar to LLAMA, which significantly outperforms Graphone. For example, it outperforms Graphone by 2.25× in terms of memory usage when the number of loaded snapshots is 27. Experiments are repeated by loading different numbers of U.K. snapshots into memory to study the memory usage. As show in Fig. 6(b), the memory usage of TgStore is slightly higher than LLAMA. However, TgStore significantly outperforms Graphone. We further conducted experiments by loading different numbers of Coauthor snapshots into memory to study the memory usage. As show in Fig. 6(c), the memory usage of TgStore is slightly lower than LLAMA. However, TgStore significantly outperforms Graphone. The different experimental results is due to the reason as follows. LLAMA adopts CSR-like data structure. Graphone adpots adjacency-list-like data structure. Like Graphone, we adopts adjacency-list-like data structure to store our snapshot union. However, there is a significant difference between TgStore and Graphone. Graphone employs pointers to recognize the objects in different snapshots. TgStore is different. Each object in the snapshot union is associated with a bitmap with bits indicating the object's presence/absence (1/0) in all snapshots.

Algorithm Execution Speed: Most of previous researches only reported the algorithm computation time in memory. However,

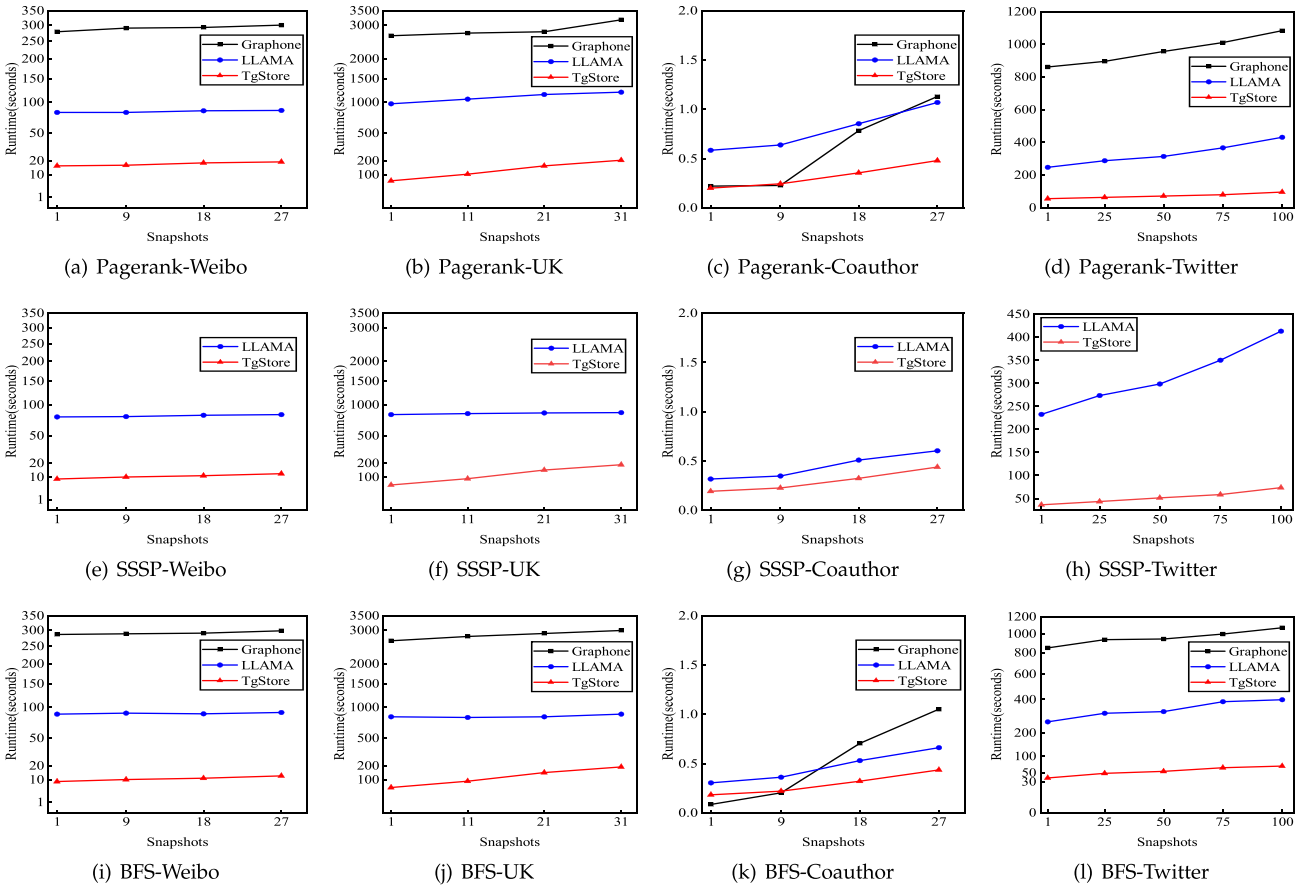


Fig. 7. Execution efficiency of algorithms.

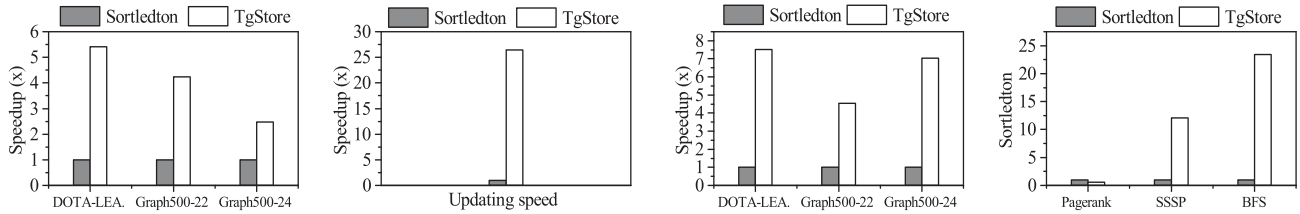
in most cases, the execution time of the graph algorithm is dominated not only by the computation but also heavily by the disk I/O time for loading and writing computation results back to disk [15]. Hence, the algorithm execution time that we discuss here includes the computation time and the disk I/O time.

Since LLAMA and Graphone can only execute one snapshot that consists of all the vertices and edges of the loaded snapshots [4], [12]. For fair comparison, TgStore also adopts this mode. Experiment is repeatedly conducted by loading Twitter with number of snapshots varying from 1 to 100. For each experiment, the Pagerank with 10 iterations are executed. Experimental results show that when executing Pagerank, the execution time is 860.03 seconds, 247.22 seconds and 54.25 seconds respectively for Graphone, LLAMA and TgStore, at the point of 1 snapshot. TgStore outperforms Graphone and LLAMA by 15.9× and 4.6× respectively. As shown in Fig. 7(d), TgStore outperforms Graphone and LLAMA by 11.5× and 4.6× respectively, at the point of 100 snapshots. Experiments are repeated by using the datasets of Weibo, U.K. and Coauthor. Like Twitter, TgStore can also outperform Graphone and LLAMA significantly, as shown in Fig. 7(a), (b), and (c).

When executing SSSP algorithm by using the dataset of Twitter, the execution time is 232.63 seconds and 36.58 seconds respectively for LLAMA and TgStore, at the point of 1 snapshot. TgStore outperforms LLAMA by 6.4×. As shown in Fig. 7(h), TgStore outperforms LLAMA by 5.6× at the point of 100 snapshots. Graphone is excluded in these experiments, since it does not provide the source code of SSSP algorithm. Experiments are repeated by using the datasets of Weibo, U.K. and Coauthor. Like Twitter, TgStore can also outperform LLAMA significantly, as shown in Fig. 7(e), (f), and (g).

When executing BFS algorithm by using Twitter dataset, the execution time is 37.43 seconds, 256.77 seconds and 848.23 seconds respectively for TgStore, LLAMA and Graphone, at the point of 1 snapshot. TgStore outperforms LLAMA and Graphone by 6.86× and 23.23× respectively. As shown in Fig. 7(l), TgStore outperforms LLAMA and Graphone by 5.88× and by 15.78× respectively, at the point of 100 snapshots. Experiments are repeated by using the datasets of Weibo, U.K. and Coauthor. Like Twitter, TgStore can outperform Graphone and LLAMA significantly, as shown in Fig. 7(i), (j), and (k). The experimental results indicate that TgStore, as an efficient storage system, can well support the time-evolving graph algorithms.

Snapshot-Level Incremental Computing: We study Tgstore in terms of the effectiveness of incremental computing. We first load 27 snapshots of Weibo into memory, and execute the Pagerank algorithm on the baseline snapshot S_0. We then execute 26 other snapshots ($S_1 \sim S_{26}$) based on the computation result of the baseline snapshot S_0. Experimental results show that, compared with baseline snapshot S_0, when executing each



(a) First snapshot creating (b) Updating (c) Loading (d) Execution of Algorithms

Fig. 8. Compared with sortledton.

TABLE III
DATASETS OF STATIC GRAPH WITH UNDIRECTED EDGES

DataSet	DOTA-LEAGUE	Graph500-22	Graph500-24
Vertices	317727	4194303	16777215
Edges	50870313	64155735	260379520

of other snapshots needs, the Pagerank algorithm reaches convergence by using the smaller number of iterations, significantly reducing the computation time. The experimental results show that incremental computing mode can obtain a speedup of 1.44 $\times \sim 2.41\times$, with an average of 1.78×. This gain stems from the efficient storage scheme of TgStore, that can load multiple snapshots into memory and then execute each snapshot independently by using the bitmaps, making the incremental computing mode to work well.

E. Compared With Sortledton

Sortledton [29] can manage a graph in external storage (such as disk) dynamically by using the lightweight relational database, SQLite. That is, it can only provide the current snapshot of the graph to the graph algorithms, since it does not store the historical snapshots of the time-evolving graph. Sortledton provides the versioning data structure in memory. However, the goal of this design is to ensure the correct results for the concurrently executing analytical queries in the presence of updates [29]. Different from Sortledton, TgStore compactly stores the snapshots of a time-evolving graph, each represents a state of the graph. Hence, we do not compare TgStore with Sortledton in terms of storage cost. We focus on the snapshot appending speed and the snapshot loading speed. As shown in Table III, experiments are conducted by using three undirected graph datasets, DOTA-LEAGUE, Graph500-22 and Graph500-24, provided by Sortledton.[6] Weibo, U.K., Coauthor and Twitter are excluded in these experiments, since Sortledton cannot support directed graphs. For fair comparison, both TgStore and Sortledton use 48 cores in each experiment.

First Snapshot Creating: We first compare TgStore with Sortledton in terms of the speed of creating the first snapshot, by using the datasets of DOTA-LEAGUE, Graph500-22 and Graph500-24. For DOTA-LEAGUE, TgStore takes 2.04 seconds to create the graph database in the disk. However, Sortledton takes 531 seconds to finish this task. Since Sortledton can only support the single-thread mode, the runtime of sortledton is divided by the number of cores (48) for the fair comparison. That is, the runtime of Sortledton is 11.06 seconds. As shown in Fig. 8(a), TgStore is 5.42× faster than Sortledton. For Graph500-22, TgStore takes 2.69 seconds to create the graph database in the disk. However, Sortledton takes 558 seconds to finish this task. Since Sortledton can only support the single-thread mode, the runtime of sortledton is divided by the number of cores (48) for the fair comparison. TgStore is 4.32× faster than Sortledton. For Graph500-24, TgStore takes 10.62 seconds to create the graph database in the disk. However, Sortledton takes 1260 seconds to finish this task. Since Sortledton can only support the single-thread mode, the runtime of sortledton is divided by the number of cores (48) for the fair comparison. TgStore is 2.47× faster than Sortledton.

Updating: We then compare TgStore with Sortledton in terms of the speed of edges updating, i.e. the updates of adding edge and deleting edge. Experiments are conducted as follows. For Graph500-22, Sortledton takes 64 minutes to finish 62×10^6 updates. The updating speed is 0.016×10^6 updates per second. We multiply the experiment result by 48 for the fair comparison. That is, the updating speed is 0.775×10^6 updates per second. Since the file format of Sortledton is different from that of TgStore and the file format of Sortledton is not been described, TgStore cannot use its updating file to finish this experiment. Hence, TgStore uses the Weibo dataset. It takes 20.38 seconds to finish 418×10^6 updates. The updating speed of TgStore is 20.51×10^6 updates per second. As shown in Fig. 8(b), TgStore is 26.46× faster than Sortledton.

Loading: We further compare TgStore with Sortledton in terms of the loading speed. Since Sortledton only stores the current snapshot of the graph, TgStore loads the last snapshot into memory. Experiments are conducted by using three datasets, DOTA-LEAGUE, Graph500-22 and Graph500-24. Experimental results show that TgStore takes 1.67 seconds to load DOTA-LEAGUE into memory. Sortledton takes 12.57 seconds to finish this task. As shown in Fig. 8(c), TgStore is 7.52× faster than Sortledton. Similarly, TgStore takes 2.43 seconds to load Graph500-22 into memory. Sortledton takes 11.02 seconds to finish this task. TgStore is 4.53× faster than Sortledton in this case. For Graph500-24, the loading time of TgStore is 9.10 seconds. Sortledton takes 64.00 seconds to finish this task. TgStore is 7.03× faster than Sortledton in this case.

[6][Online]. Available: https://zenodo.org/records/3966439

Execution Efficiency of Algorithms. We further compare TgStore with Sortledton in terms of the execution efficiency of algorithms. Since Sortledton can only provide the current snapshot of the graph, TgStore executes the last snapshot of the time-evolving graph. Experiments are conducted by using the algorithms Pagerank, SSSP and BFS, on the DOTA-LEAGUE dataset. For each experiment, the Pagerank with 10 iterations are executed. For the Pagerank algorithm, the runtime of TgStore is 0.341 seconds. Sortledton takes 0.206 seconds to finish this task. As shown in Fig. 8(d), Sortledton is 1.65× faster than TgStore. For the SSSP algorithm, the runtime of TgStore is 0.045 seconds. Sortledton takes 0.545 seconds to finish this task. TgStore is 12.11× faster than Sortledton in this case. For the BFS algorithm, the runtime of TgStore is 0.011 seconds. Sortledton takes 0.258 seconds to finish this task. TgStore is 23.45× faster than Sortledton in this case. These experimental results show that Sortledton has a higher execution efficiency when executing the task of graph analytical workload. However, TgStore has a higher execution efficiency in cases of graph traversal workload.

V. Related Work

In this section, we briefly discuss the storage methods of time-evolving graph that are not mentioned in Section II.

Timestamp is a known method to store time-evolving graph by recording the timestamps of each vertex or edge with the time evolving. However, additional storage for storing the timestamp will double the storage cost [30]. It also introduces the additional complexity to reconstruct snapshots, leading to performance degradation. Furthermore, adjacency list format is regularly employed to support the timestamp method. It is inefficient to maintain the adjacency list at edge-level granularity [30].

Graphone [4], LLama [12] and Teseo [31] use read and write-optimized segments to handle inserts. This method will lead to lower read performance or reduced freshness. For LLama and GraphOne, this method will result in unstable throughput over time. Different from them, TgStore can reconstruct any snapshots by using the snapshot union set (S∪) and the bitmaps efficiently. It can also append new snapshots rapidly by using the well-designed data structure of Structure-Changing Graph Representation (SCGR) that avoids data removement during the process of snapshot appending. LLama and GraphOne ingest all buffered writes into read-optimized storage. Similarly, Teseo creates a snapshot in $O(V)$ steps before starting analytics. GraphOne stores the sizes of the neighborhoods to guarantee isolation from new updates. TgStore executes the task of snapshot appending in memory. When a new logfile arrives, it loads the FEDB, NEDBs and the logfile into the memory together. It separates the edge bitmaps from the FEDB and NEDBs by using edge bitmap files that are also loaded into the memory. This design improves the snapshot appending speed significantly. Since we only need to load FEDB and NEDBs from disk to build the indexes that help find the bitmap position of an added or deleted edge.

As a graph storage method, it is important to support the updates (adding, deleting and updating of vertex and edge). OrientDB [32] builds a graph model using a document store model where all information of a vertex, including all of its neighbors, are stored in a document. This method is inefficient for the updates of vertices or edges. For example, the size of invalid storage space increases when deleting the vertices and edges. It is time-consuming to recycle the invalid storage space by moving the data. In order to overcome these problems, we first design the snapshot union set with edge bitmaps to represent a time-evolving graph, significantly reducing the storage cost of average snapshot. We then design the data structure for the snapshot union set with edge bitmaps carefully, which can avoid data moving and rebuilding, leading to the high speed of snapshot appending.

Another recent work, Sortledton [29] is a universal, transactional graph data structure. It can provide a high speed of transactional updates by carefully designing an adjacency list-like universal data structure, while supporting the high performance execution of heterogeneous graph workloads. However, when processing graph workloads on dynamic datasets, Sortledton [29] is slower than running the computations on a static graph stored in CSR [33] format. Like other specialized systems, TgStore focuses mainly on the disk storage efficiency of time-evolving graph. For in-memory layout, TgStore allocates the appropriate size for each vertex's neighbor array based on indexes and bitmaps, without reserving extra space. Unlike TgStore, Sortledton adopts the method of reserving space, reducing memory utilization. Sortledton edge versions record operation types (insertion, update, or deletion), submission time, and related attributes. When accessing graphs at different moments, the graph needs to be reconstructed according to related edge information, resulting in additional overhead. By using the snapshot union set (S∪) with the bitmaps, TgStore stores time-evolving graphs with a very low storage overhead and reconstructs snapshots at a high speed.

Ck^d-tree [34] represents temporal graphs as cells in a 4D binary matrix: two dimensions to represent extreme vertices of an edge and two dimensions to represent the temporal interval when the edge exists. It is capable of dealing with unclustered data with a good use of space. However, tree-based techniques can also reduce the performance when accessing the edges (leaves). In order to locate one edge, it needs to access the intermediate nodes between the root node and the edge node. Nelson et al. [35] proposed a compressed data structure that has a compressed binary tree corresponding to each row of each adjacency matrix of the time-evolving graph. Brisaboa et al. [36] explore Compressed Suffix Array (CSA), a well-known compact and self-indexed data structure in the area of text indexing, to represent large temporal graphs. This method uses a reasonable space and is efficient for solving complex temporal queries. The methods above are efficient for the tasks of graph query where only a few inconsecutive vertices and edges need to be accessed. However, they are inefficient for the graph analysis tasks where almost all the vertices are active and need to be accessed. For the graph analysis task, it is efficient to load the whole snapshot of the time-evolving graph into the memory. TgStore is design to effectively store the snapshots of the time-evolving graph, which aims to efficiently support the time-evolving graph analysis tasks. We design the union of all snapshots with one

bitmap for each object (vertex or edge) to avoid the repeated objects among snapshot of the time-evolving graph. We do not compress the graph structure union of all snapshots, so that each snapshot can be accessed efficiently. Furthermore, the union of all snapshots has a low enough storage cost for a time-evolving graph.

VI. Conclusion

We have presented TgStore, an efficient storage system for large time-evolving graphs, that has a low storage cost of average snapshot with the high snapshot access speed. As a storage system, TgStore can well support the time-evolving graph algorithms by efficiently providing the snapshots that the algorithm needs, significantly reducing the algorithm execution time and the needed memory space.

As future work, we first plan to extend TgStore to store the changing graph in a continuous way. It can not only store the historical snapshots but also support the dynamic updates in a continuous way. We then plan to design an efficient compression method for the bitmaps. Since the storage cost of the bitmaps will increase if the external business system needs to add snapshots frequently. TgStore stores a time-evolving graph by using the snapshot union set ($S\cup$) with one bitmap for each object (vertex or edge). The snapshot union set eliminates all the repeated objects among the snapshots. Hence, the size of the snapshot union set does not increase rapidly when TgStore creates a new snapshot frequently. The challenge is the storage cost for the bitmaps. In order to access each snapshot efficiently, the bitmap of each object in TgStore has the same size, making it easy to determine if an object belongs to a certain snapshot or not. Hence, an efficient compression method is required, which can not only reduce the storage cost of the bitmaps significantly but also guarantee that each snapshot can be accessed efficiently.

References

[1] G. Malewicz et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.

[2] A. Guo, Q. Zhang, and X. Zhao, "A Graph-Based Approach Towards Risk Alerting for COVID-19 Spread," in *Proceedings of Software Foundations for Data Interoperability and Large Scale Graph Data Analytics*. Berlin, Germany: Springer, 2020, pp. 63–69.

[3] W. Han et al., "Chronos: A graph engine for temporal graph analysis," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–14.

[4] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," *ACM Trans. Storage*, vol. 15, no. 4, pp. 1–40, 2020.

[5] W. Huo and V. J. Tsotras, "Efficient temporal shortest path queries on evolving social graphs," in *Proc. 26th Int. Conf. Sci. Stat. Database Manage.*, 2014, pp. 1–4.

[6] N. Martínez-Bazan, M. Á. Águila-Lorente, and V. Muntés-Mulero, "Efficient graph management based on bitmap indices," in *Proc. 16th Int. Database Eng. Appl. Sysmp.*, 2012, pp. 110–119.

[7] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2013, pp. 135–146.

[8] M. Bender and H. Hu, "An adaptive packed-memory array," *ACM Trans. Database Syst.*, vol. 32, no. 4, pp. 26–40, 2007.

[9] G. Koloniari, D. Souravlias, and E. Pitoura, "On graph deltas for historical queries," 2013, *arXiv:1302.5549*.

[10] G. Koloniari and E. Pitoura, "Partial view selection for evolving social graphs," in *Proc. 1st Int. Workshop Graph Data Manage. Experiences Syst.*, 2013, pp. 1–6.

[11] U. Khurana and A. Deshpande, "Efficient snapshot retrieval over historical graph data," in *Proc. IEEE 29th Int. Conf. Data Eng.*, 2013, pp. 997–1008.

[12] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 363–374.

[13] Y. Cheng et al., "A highly cost-effective task scheduling strategy for very large graph computation," *Future Gener. Comput. Syst.*, vol. 89, pp. 698–712, 2018.

[14] Y. Cheng, H. Jiang, F. Wang, Y. Hua, and D. Feng, "Using high-bandwidth networks efficiently for fast graph computation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 5, pp. 1170–1183, May 2019.

[15] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 31–46.

[16] Y. Yang, J. X. Yu, H. Gao, J. Pei, and J. Li, "Mining most frequently changing component in evolving graphs," *World Wide Web*, vol. 17, no. 3, pp. 351–376, 2014.

[17] R. A. Rossi, B. Gallagher, J. Neville, and K. Henderson, "Modeling dynamic behavior in large evolving graphs," in *Proc. 6th ACM Int. Conf. Web Search Data Mining*, 2013, pp. 667–676.

[18] M. A. Sakr and R. H. Güting, "Group spatiotemporal pattern queries," *GeoInformatica*, vol. 18, no. 4, pp. 699–746, 2014.

[19] A. G. Labouseur et al., "The G* graph database: Efficiently managing large distributed dynamic graphs," *Distrib. Parallel Databases*, vol. 33, no. 4, pp. 479–514, 2015.

[20] Z. Cai, D. Logothetis, and G. Siganos, "Facilitating real-time graph mining," in *Proc. 4th Int. Workshop Cloud Data Manage.*, 2012, pp. 1–8.

[21] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, and M. Wu, "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 85–98.

[22] V. Z. Moffitt and J. Stoyanovich, "Towards a distributed infrastructure for evolving graph analytics," in *Proc. 25th Int. Conf. Companion World Wide Web*, 2016, pp. 843–848.

[23] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *Proc. 20th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2015, pp. 183–193.

[24] X. Xu, F. Wang, H. Jiang, Y. Cheng, D. Feng, and Y. Zhang, "A hybrid update strategy for I/O-efficient out-of-core graph processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 8, pp. 1767–1782, Aug. 2020.

[25] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," 1999.

[26] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proc. VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.

[27] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?," in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 591–600.

[28] Y. Pan, R. Pearce, and J. D. Owens, "Scalable breadth-first search on a GPU cluster," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2018, pp. 1090–1101.

[29] P. Fuchs, D. Margan, and J. Giceva, "Sortledton: A universal, transactional graph data structure," *Proc. VLDB Endowment*, vol. 15, no. 6, pp. 1173–1186, 2022.

[30] M. Then, T. Kersten, S. Günnemann, A. Kemper, and T. Neumann, "Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs," *Proc. VLDB Endowment*, vol. 10, no. 8, pp. 877–888, 2017.

[31] D. De Leo and P. Boncz, "Teseo and the analysis of structural dynamic graphs," *Proc. VLDB Endowment*, vol. 14, no. 6, pp. 1053–1066, 2021.

[32] O. Database, "Retrieved from,"2016. [Online]. Available: https://orientdb.org/

[33] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: SIAM, 2003.

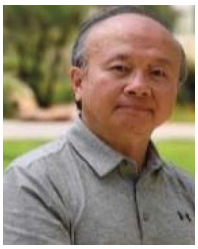
[34] D. Caro, M. Rodrguez, N. R. Brisaboa, and A. Farina, "Compressed k-tree for temporal graphs," *Knowl. Inf. Syst.*, vol. 49, pp. 553–595, 2015.

[35] M. Nelson, S. Radhakrishnan, C. Sekharan, A. Chatterjee, and S. G. Krishna, "Queryable compression on time-evolving web and social networks with streaming," *ACM Trans. Web*, vol. 16, no. 2, pp. 1–21, 2021.

[36] N. R. Brisaboa, D. Caro, A. Farina, and M. A. Rodriguez, "Using compressed suffix-arrays for a compact representation of temporal-graphs," *Inf. Sci.*, vol. 465, pp. 459–483, 2018.

Yongli Cheng (Member, IEEE) received the BE degree from the Chang'an University, Xi'an, China, in 1998, the MS degree from the Fuzhou University, Fuzhou, China, in 2010, and the PhD degree from the Huazhong University of Science and Technology, Wuhan, China, 2017. He is currently a teacher with the College of Computer and Data Science, Fuzhou University currently. His current research interests include computer architecture and graph computing. He has more than 20 publications in major international conferences and journals, including INFOCOMM, HPDC, ICPP, IWQoS, the *Transactions on Parallel and Distributed Systems*, *IEEE/ACM Transactions on Networking*, *Future Generation Computing Systems*, and *Frontiers of Computer Science*.

Yan Ma received the BS degree from the Huaqiao University, Quanzhou, China, in 2019. He is currently working toward the MS degree majoring in computer technology with Fuzhou University, Fuzhou, China. His current research interests include time-evolving graph storage techniques and applications.



Hong Jiang (Fellow, IEEE) received the BE degree from the Huazhong University of Science and Technology, Wuhan, China, in 1982, the MASc degree from the University of Toronto, Toronto, Canada, in 1987, and the PhD degree from the Texas A&M University, College Station, Texas, in 1991. He is Wendell H. Nedderman endowed professor and chair with the Department of Computer Science and Engineering, University of Texas at Arlington. His research interests include computer architecture, computer storage systems, and parallel/ distributed computing. He serves as an associate editor of *IEEE Transactions on Parallel and Distributed Systems*. He has more than 200 publications in major journals and international Conferences in these areas, including *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *ACM Transactions on Storage*, *ACM Transactions on Architecture and Code Optimization*, *Journal of Parallel and Distributed Computing*, ISCA, MICRO, FAST, USENIX ATC, USENIX LISA, SIGMETRICS, MIDDLEWARE, ICDCS, IPDPS, OOPLAS, ECOOP, SC, ICS, HPDC, and ICPP.

Lingfang Zeng (Member, IEEE) received the BS degree in computer application from the Huazhong University of Science and Technology, Wuhan, China in 2000, the MS degree in computer application from China the University of Geoscience, China, in 2003, and the PhD degree in computer architecture from HUST, in 2006. He was research fellow with the Department of Electrical and Computer Engineering, National University of Singapore, Singapore, during 2007-2008 and 2010-2013, and a visiting professor with Johannes Gutenberg University Mainz, Germany (2016-2018). He is currently with Zhejiang Lab as a PI. He publishes more than 60 papers in major journals and conferences.

Fang Wang (Member, IEEE) received the BE and master's degrees in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 1994 and 1997, respectively, and the PhD degree in computer architecture from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2001, where she is currently a professor in computer science and engineering. Her research interests include distribute file systems, parallel I/O storage systems, and graph processing systems. She has more than 50 publications in major journals and conferences, including the Future Generation Computing Systems, the *ACM Transactions on Architecture* and Code Optimization, HiPC, ICDCS, HPDC, and ICPP.

Xianghao Xu (Member, IEEE) received the PhD degree from the Huazhong University of Science and Technology, Wuhan, China, 2021. He is currently an Assistant Professor with the School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, China. His current research interests include graph processing, computer architecture and Big Data analytics. He has several publications in major international conferences and journals, including *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Big Data*, ICPP, IWQoS and FCS.

Yuhang Wu received the BE degree from the Fujian Normal University, Fuzhou, China, in 2019. He is currently working toward the MS degree majoring in computer technology with FuZhou University, FuZhou, China. His current research interest includes graph processing.