An Efficient Snapshot Strategy for Dynamic Graph Storage Systems to Support Historical Queries

LUO XIANGYU1,2, LUO YINGXIAO1, GUI XIAOLIN2, AND YU ZHENHUA1

1College of Computer Science and Technology, Institute of Systems Security and Control, Xi’an University of Science and Technology, Xi’an 710054, China
2School of Electronics and Information Engineering, Xi’an Jiaotong University, Xi’an 710049, China

Corresponding author: Yu Zhenhua (zhenhua_yu@163.com)

This work was supported in part by the National Natural Science Foundation of China under Grant 61702408 and Grant 51634007, in part by the Natural Science Foundation of Shaanxi Province, China, under Grant 2019JM-020 and Grant 2019GY-056, and in part by the Innovation Group for Interdisciplinary Computing Technologies, Xi’an University of Science and Technology.

ABSTRACT Large-scale dynamic graphs typically involve big data. Recently a dynamic graph storage system is required to be capable of recreating any historical state to support historical queries. A typical storage solution supporting historical queries is called ‘snapshot plus log’. A snapshot records the whole data at a certain moment, while the log file is responsible for saving all the update operations. The historical state is then recreated from the nearest snapshot by redoing or undoing the related update operations saved in the log file. The challenge lies in how to minimize both the number of snapshots and that of redone and undone operations performed in historical state recreation. The traditional system stores snapshots at regular intervals. However, historical states do not share the same frequency of being requested. Therefore, the traditional strategy is very inefficient. This paper proposes a new strategy that determines the timestamps of the snapshots based on the distribution of the historical queries. First, the historical queries are clustered into a given number of groups according to the timestamps of the requested historical states, and the cluster centroids are calculated. Second, the snapshots are created according to the timestamps of the cluster centroids. Since the cluster centroids may change as time goes by, the above process is executed periodically. Experimental results show that with the same storage costs, the snapshot strategy proposed in this paper greatly improves the performance of recreating historical states, leading to at least 70.7% computation reduction in terms of the number of both redone and undone operations. Besides, with the same recreation performance guarantee, it brings nearly 78.9% storage reduction on average in terms of the number of snapshots.

INDEX TERMS Dynamic graph, snapshot, log, clustering method, historical query.

I. INTRODUCTION

As an important part of big data, dynamic graph data are widespread in many applications such as social networks [1]–[3], communication networks [4], [5], biology and disease networks [6], [7], coauthor networks [8], [9], etc. Graph is also a useful tool in areas such as manufacturing [10]. Recently people not only make analysis of the real time state of a graph, but also pay attention to how a graph evolves to obtain more knowledge [11], [12]. For example, Leskovec et al. [13] studied diameter changes of an evolving social network. Semertzidis and Pitoura [14] investigated the most durable graph pattern that exists for the longest period of time.

The increasing demands for evolving analysis pose great challenges to the dynamic graph storage system. The system is required to be capable of handling historical queries. In other words, it should be able to recreate any historical states [15], [16].

There are three optional storage models for dynamic graph storage systems to support historical queries. The first is called ‘sequence of snapshots’. A dynamic graph is regarded as a sequence of snapshots. Each snapshot is a static graph representing the state of the dynamic graph at a certain moment. The second is ‘log file’. All the updates of a dynamic graph are saved in a log file. Therefore, any historical state of the graph can be regenerated through redoing the update
operations recorded in the log file. The third is named ‘set of varying instances’. A dynamic graph is regarded as a set of dynamically changing instances. Each instance contains the information about the evolving process of one single vertex, e.g., when the vertex is created, how and when the attribute of the vertex is updated, and how and when its relationships with other vertices change.

The first model enables high recreation performance at the cost of very high storage consumption, while the second one consumes least storage space suffering from poor recreation performance. The third one saves storage consumption as well as provides high performance in recreating the historical state of a single vertex or a small sub-graph. However, it is inefficient to recreate the historical state of the whole graph [17].

The combination the first two models named ‘snapshot plus log’ has been investigated in the area of fault tolerance [18]. It provides storage consumption as well as high performance in state recovery. In the scenario, snapshots are stored at regular intervals. Besides, since only the latest state is required to be recovered, the older snapshots could be discarded to reduce storage consumption as time goes by.

We investigate how to apply this ‘snapshot plus log’ model in the dynamic graph storage system for supporting historical queries. The main challenge lies in the snapshot strategy. The traditional one that stores snapshots at regular intervals is inefficient in this new situation, since the historical states do not share the same frequency of being requested. With the traditional snapshot strategy, some snapshots may be seldom accessed by any historical queries, while some historical queries may request historical states not near to any snapshot. The former case leads to waste of storage space, while the latter case leads to recreation performance degradation.

The contribution of this paper includes three aspects. First, we formally define the snapshot optimization problem. It is stated as the minimization of the number of redone and undone operations in the historical state recreation process through the optimal selection of the timestamps for the snapshots. Second, a new snapshot strategy is proposed to solve the problem. The historical queries are clustered according to the timestamps of the requested historical states and the centroids are used to determine the location of the snapshots. The novelty lies in that the distribution of the snapshots is not uniform in the time axis but consistent with the density of the historical queries. Last but not least, we conduct experimental analysis to validate the efficiency of the proposed strategy. The results show that with the same number of snapshots, the proposed strategy greatly improves the performance of historical state recreation. Meanwhile, with the same creation performance guarantee, the proposed strategy sharply reduces the storage consumption.

The paper is organized as follows. Section II summarizes the related work; Section III formally defines and analyzes the problem; Section IV elaborates upon the proposed snapshot strategy; Section V shows the experimental results; Section VI concludes the whole paper.

II. RELATED WORK

Recently the research on the storage of dynamic graphs with support for historical queries attracts much attention [19], [20]. The fundamental challenge here is the tradeoff between recreation performance and storage consumption [21]. The storage model can be classified into the following three categories: the snapshot sequence model, the log file model and the set of varying instances model.

The snapshot sequence model regards the evolution of a dynamic graph as a set of static graph snapshots. One graph snapshot represents the state of the graph at a certain moment. This approach may be suitable for small-size and slowly evolving graphs [22]. Both Yang et al. [23] and Ren et al. [24] adopted the snapshot sequence model for historical analysis. To improve space efficiency, Ren et al. [24] used a compression method to reduce data redundancy at the cost of update time increase. Zaki et al. [25] proposed a method for storing the sequence of snapshots in a compact manner while maintaining a very low update overhead. The main idea is to separate the parts of the data that may change in the future from other parts. For a given set of graph snapshots which correspond to the state of an evolving graph at different time instances, Semertzidis et al. [26] discussed the Best Friends Forever (BFF) problem, i.e., how to identify the set of nodes that are the most densely connected in all snapshots. Nelson et al. [27] investigated several problems of interest on time-evolving graphs and proposed corresponding algorithms that run on compressed time-evolving graphs. The advantage of the methods based on the snapshot sequence model is that the requested historical state can be quickly obtained without any recreation cost, while the disadvantage is that the storage consumption is too high.

The log file model saves all the update operations of a dynamic graph in a log file. Theoretically, all the historical states can be recreated by redoing the update operations, and the storage consumption is the minimum. However, it is not feasible in practice, since it takes too long to recreate a historical state.

In the third storage model, a dynamic graph is regarded as a set of varying instances. Each instance contains the update history of a single vertex [17], [28]. Han et al. [29] proposed a method that divides the graph into multiple spatio-temporal chunks with each chunk covering a subset of vertices and spanning a certain time interval. The method is suitable for graph state recreation in a distributed manner. Aridhi et al. [30] also discussed the distributed processing of large scale dynamic graphs. The advantage of this model lies in the efficient historical query of a single vertex or a small sub-graph, but the disadvantage is the inconvenience for recreating the whole historical state for a given timestamp.

In order to make a tradeoff between the performance of historical state recreation and the storage reduction, some researchers try to combine the first two models [31]. It is called as ‘snapshot plus log’. The main idea is to save the state of the graph as a snapshot at some moments, and record the update operations such as vertex or edge addition, deletion,
and attribute modification in a log file. When a historical query arrives, the snapshot nearest to the requested historical state is retrieved, and the update operations happened between the requested historical state and its nearest snapshot will be redone or undone depending on whether the snapshot is earlier or later than the requested historical state. After redoing or undoing the update operations, the historical state will be recreated. Bhattacherjee et al. [21] also discussed the tradeoff between recreation performance and storage cost. However, the method is designed for handling multiple-versioned data, not for dynamically changing data.

In the ‘snapshot plus log’ storage model, it is difficult to determine when to save a snapshot. The objective is to store a minimum number of snapshots while maintaining a small number of redone and undone operations in the recreation process, or minimizing the number of redone and undone operations for a given number of snapshots. At present, there are two typical snapshot strategies. One is based on time interval, and the other is based on update operation. The first stores snapshots at regular intervals, while the second stores a snapshot whenever the number of the update operations accumulates to a certain value. However, whether the snapshots are appropriate depends on whether they hit the historical queries, or at least near to the requested historical states. Snapshots far away from the historical states do not favor the recreation process at all while consuming storage space.

III. PROBLEM STATEMENT

We use \( U \) as the set of the update operations that have happened on the dynamic graph \( G \), with each element \( u_i \) representing the \( i \)th update operation. Here each \( u_i \) is a tuple consisting of \( sn_i, ut_i \) and \( o_i \), with \( sn_i \) denoting the serial number of the update operation, \( ut_i \) representing the time when the update operation \( u_i \) happened and \( o_i \) describing how the graph \( G \) was updated at time \( ut_i \). Let \( M \) denote the number of elements contained in \( U \), and then there are totally \( M \) different historical states. The valid timestamps corresponding to the \( i \)th historical state can be described as an interval \( [ut_i, ut_{i+1}) \). It means that at every time instant during the time interval \( [ut_i, ut_{i+1}) \), the state of the graph \( G \) is always the same to that at time \( ut_i \).

We use \( Q \) as the set of historical queries that are predicted to happen in the near future, with the element \( q_j \) representing the \( j \)th kind of historical query. Here each \( q_j \) is a tuple consisting of \( ht_j \) and \( n_j \), with \( ht_j \) representing the timestamp related to the requested historical state of the graph \( G \) and \( n_j \) representing the frequency of the \( j \)th kind of historical query in the near future. In other words, the historical state of the graph \( G \) at time \( ht_j \) will be queried \( n_j \) times. It should be noted that, \( Q \) is a predicted set, and the more exact it is, the better. A simple prediction method is to take the latest query set as the predicted value. The prediction technique is out of the scope of this paper. We just assume that \( Q \) is available through some prediction methods.

We use \( S \) as the set of the snapshots and \( T \) as the set of the timestamps corresponding to the snapshots in \( S \). To serve a historical query that requests the historical state of the graph \( G \) with the timestamp \( ht_j \), the system works as follows. First, it finds the two neighboring snapshots \( s_{left} \) and \( s_{right} \). Let \( s_{left} \) and \( s_{right} \) respectively represent the timestamps of the two neighboring snapshots. We obtain that \( s_{left} \) is the largest snapshot timestamp in those not greater than \( ht_j \), and \( s_{right} \) is the smallest snapshot timestamp in those not smaller than \( ht_j \). Second, the system calculates the distances from the requested historical state to the two neighboring snapshots. The nearer neighboring snapshot is chosen for further computation. Here the distance between a historical state and a snapshot is evaluated by not the time interval itself, but the number of the update operations occurred during the time interval. Finally, the requested historical state is recreated by redoing or undoing operations on the nearer snapshot. If \( s_{left} \) is the nearer snapshot, the update operations during the interval \( (s_{left}, ht_j) \) need to be redone and the process is called forward recreation. Otherwise, the update operations during the time interval \( (ht_j, s_{right}] \) need to be undone and the process is called backward recreation. The update operation details could be looked up in the log file that records the whole updating process of the graph. The number of the update operations that are redone and undone greatly affects the performance of historical queries.

Constrained by the storage cost, we assume that only \( N \) snapshots of the graph \( G \) are allowed to be created. The problem is how to determine \( T \) for the \( N \) snapshots such that the average number of redone and undone operations served for the historical queries is minimized. Suppose that \( ts_k (1 \leq k \leq N) \) represents the timestamp of the \( k \)th snapshot. The optimization problem is then stated as follows:

\[
\begin{align*}
l(j) &= \max_{ts_k \leq ht_j} \{ts_k\}, \quad j = 1, \ldots, |Q| \\
r(j) &= \min_{ts_k \geq ht_j} \{ts_k\}, \quad j = 1, \ldots, |Q| \\
\text{sn}(t) &= \max_{ut_i \leq t} \{sn_i\} \\
O(j) &= \min\{\text{sn}(ht_j) - \text{sn}(l(j)), \text{sn}(r(j)) - \text{sn}(ht_j)\} \\
f(ts_1, \ldots, ts_N) &= \sum_{j=1}^{|Q|} (O^2(j) \ast n_j) / \sum_{j=1}^{|Q|} n_j \\
\text{min}\{f(ts_1, \ldots, ts_N)\}
\end{align*}
\]

In Equation (5), we use \( O^2(j) \) rather than \( O(j) \), because the former promises lower variation.

IV. THE PROPOSED SNAPSHOT STRATEGY

A. MAIN IDEA

The two traditional snapshot strategies (i.e., the time interval based strategy and the update operation based strategy) make decisions independent of the distribution of historical queries. As a result, some snapshots are far away from all the historical states requested by the historical queries, while some historical queries request historical states that are not near to any
snapshot. The former case leads to storage waste, while the latter case leads to recreation performance degradation.

Therefore, we propose a new snapshot strategy that determines the timestamps of the snapshots according to the distribution of historical queries. Firstly, a clustering method is used to aggregate the historical queries into a number of groups. The number of the groups equals the number of the snapshots to store. Secondly, the cluster centroids are used to determine the timestamps of the snapshots. Historical queries aggregated to the same group will be served by the same snapshot. Some of the historical queries are served in a forward recreation manner by redoing update operations on the snapshot, while others are served in a backward recreation manner by undoing update operations on it.

To make a further explanation of the idea of our snapshot strategy, we take an example as shown in FIGURE 1. In the figure, the graph has been updated 10 times during the period from the time instant 0 to 20, and the update operations are represented with \( u_1 \) . . . \( u_{10} \). There are eight historical queries, i.e., \( h_1 \) . . . \( h_8 \), and the timestamps of the requested historical states are marked in the figure. Suppose that only two snapshots are allowed to be stored.

With the time based snapshot strategy, the timestamps of the two snapshots are 10 and 20 respectively. All of the historical states requested by the eight historical queries are created from Snapshot1. For serving the historical query either \( h_1 \) or \( h_2 \), four update operations, i.e., \( u_3 \), \( u_4 \), \( u_5 \), and \( u_6 \), need to be undone. For serving either \( h_3 \) or \( h_4 \), three update operations, i.e., \( u_4 \), \( u_5 \), and \( u_6 \), need to be undone. For serving either \( h_5 \) or \( h_6 \), no operation needs to be undone or redone, because the requested historical state is equal to Snapshot1. For serving either \( h_7 \) or \( h_8 \), one update operation, i.e., \( u_7 \), needs to be redone. Therefore, the average number of redone or undone operations for serving each historical query equals 2.

With the update operation based snapshot strategy, the first snapshot is created once the number of the update operations reaches five, and so the timestamps of the two snapshots are six and twenty respectively. All of the historical states requested by the eight historical queries are created from Snapshot1. For serving the historical query either \( h_1 \) or \( h_2 \), three update operations, i.e., \( u_3 \), \( u_4 \), and \( u_5 \), need to be undone. For serving either \( h_3 \) or \( h_4 \), two update operations, i.e., \( u_4 \) and \( u_5 \), need to be undone. For serving either \( h_5 \) or \( h_6 \), one operation, i.e., \( u_6 \), needs to be redone. For serving either \( h_7 \) or \( h_8 \), two update operations, i.e., \( u_6 \) and \( u_7 \), need to be redone. Therefore, the average number of redone or undone operations for serving each historical query also equals 2.

With our cluster based snapshot strategy, the two snapshots are located at the centroids of the historical queries, and the two timestamps are four and eleven. For the two historical queries \( h_1 \) and \( h_2 \), the requested historical states are recreated from Snapshot1 by undoing the update operation, i.e., \( u_3 \). For the two historical queries \( h_3 \) and \( h_4 \), the requested historical states are equal to Snapshot1 without redoing or undoing any update operation. For the two historical queries \( h_5 \) and \( h_6 \), the requested historical states are recreated from Snapshot2 by undoing the update operation, i.e., \( u_7 \). For the two historical queries \( h_7 \) and \( h_8 \), the requested historical states are equal to Snapshot2 without redoing or undoing any update operation. Therefore, the average number of redone or undone operations for serving each historical query equals 0.5.

Our cluster based snapshot strategy ensures that the historical queries are served by a very near snapshot with a very high probability, and the number of redone and undone operations is minimized in recreating the requested historical states.
B. ELABORATIONS

As stated in the above subsection, there are two phases in our strategy for determining the snapshot timestamps. One is to cluster the historical queries, and the other is to obtain the timestamps of the snapshots through the cluster centroids. We elaborate upon the two phases in the following.

1) PHASE 1: CLUSTERING THE HISTORICAL QUERIES

We use the K-means algorithm for clustering historical queries. Although other clustering algorithms may also work, we verify that the simple K-means algorithm already meets our requirements in the situation, as shown in Section V.

The K-means algorithm clusters the samples in an iteration way. It initializes the centroids with random values. In each iteration, there are three tasks. The first is to calculate the distance between each sample and each centroid. The second is to classify the samples into groups. According to the values of the distances calculated in the first step, the nearest centroid for each sample is figured out, and samples with the same nearest centroid are classified into the same group. The third is to update the centroids according to the samples in the group, and start the next iteration step.

In the K-means algorithm, a central part is to calculate the distance between a given sample and a given centroid. In our situation, the samples are the historical queries, and the centroids are the timestamps of the snapshots. A simple indicator for the distance is the time interval between the timestamp of the historical state requested by the historical query and the timestamp of the snapshot. However, the time interval length but the number of update operations occurred during the time interval actually matters, as it determines the cost of redoing and undoing in the historical state recreation. Therefore, we adopt the number of update operations occurred during the time interval between the timestamp of the requested historical state and that of the snapshot as the criterion of the distance.

For example, in FIGURE 2, the time interval between the timestamp of the requested historical state and that of Centroid1 is 4, while the time interval between the timestamp of the requested historical state and that of Centroid2 is 6. However, the distance between the requested historical state and Centroid1 is longer than that between the requested historical state and Centroid2. The reason is that there are three update operations during the former time interval, while two update operations during the latter.

For the convenience of calculating the distance defined above, we take the serial number of the latest update operation to substitute the timestamp for describing the historical state. Suppose that the timestamp of the requested historical state is \(ht_i\). Among all the update operations in \(U\), we look for \(ut_i\) satisfying that the element \(ut_i\) is the maximum one among those not greater than \(ht_i\), and then \(sn_i\) is the serial number of the latest update operation. In the clustering method, we use \(sn_i\) to denote the requested historical state requested by \(q_i\).

The historical query clustering algorithm is shown in Algorithm 1. The input of the algorithm includes the update operation set \(U\), the historical operation set \(Q\), and the number of snapshots to be stored \(N\), while the output \(C\) is the set of cluster centroids in the form of the sequential numbers of the update operations. Each element \(c_k\) in \(C\) will be transformed into snapshot timestamps in Phase 2. Here we just describe the transformation as \(ts_k = g(c_k)\).

From Lines 1 to 11, the algorithm transforms the historical query set from \(Q\) to \(Q'\). The timestamp of a requested historical state is substituted by the serial number of the latest update operation. As stated above, it is more convenient to calculate the distance in this way. From Lines 12 to 13, the cluster centroids are initialized with random values, and they are optimized in an iteration way as shown from Lines 14 to 25. In each iteration, each historical request \(q'_i\) is attributed to the nearest centroid as shown from Lines 17 to 20, and then the cluster centroids are updated according to the belonging historical queries as shown from Lines 21 to 24. Besides, the cluster centroids are saved as \(C_{old}\) at the beginning of each iteration shown in Line 15, and are compared with the newly updated cluster centroids \(C\) at the end of each iteration shown in Line 25. If there is no difference between \(C\) and \(C_{old}\), the iteration process is finished, and the cluster centroids are returned.

In each iteration, each \(c_k\) is updated with the average of the sequential numbers of the latest update operations with regard to the historical states requested by the historical queries belonging to the cluster. The reason is explained below.

\[
f(ts_1, \ldots, ts_N) = f(g(c_1), \ldots, g(c_N)) = \sum_{k=1}^{N} \sum_{q_i \in \text{Cluster}_k} (sn_i - c_k)^2 / \sum_{j=1}^{\left|Q\right|} n_j \quad (7)
\]

\[
\frac{\partial f}{\partial c_k} = 2\sum_{q_i \in \text{Cluster}_k} sn_i - |\text{Cluster}_k|c_k / \sum_{j=1}^{\left|Q\right|} n_j \quad (8)
\]
To minimize \( f \), Equation (8) should equal 0, and we obtain

\[
c_k = \frac{\sum_{q_i \in C_k} sn_i}{|C_k|} = \text{avg}_k
\]  

(9)

2) PHASE 2: CALCULATING THE SNAPSHOT TIMESTAMPs FROM THE CLUSTER CENTROIDS

The centroids obtained in Phase 1 are described with the serial numbers of the update operations, and we need to convert them to snapshot timestamps.

First, according to the serial number, the time instant when the update operation occurred can be obtained from the information contained in the set \( U \). Second, the time instant when the following update operation occurred can also be obtained. Any time between the two time instants could be taken as the timestamp of the snapshot. Without loss of generality, we take the first time instant as the snapshot timestamp.

The calculation of the snapshot timestamps based on the cluster centroids is shown in Algorithm 2.

Since the centroids of historical queries may change as time goes by, the timestamps of the snapshots will be updated periodically. The snapshots can be created according to their timestamps.

The serving of the historical queries is the same to other traditional strategies. First, according to the timestamp of the requested historical state, two nearest snapshots are found, i.e., the left nearest snapshot and the right nearest snapshot. Second, the number of update operations between the timestamp of the left nearest snapshot and the timestamp of the requested historical state is obtained, and that between the right nearest snapshot and the historical state is also obtained. Third, according to the number of update operations, the real nearest snapshot is determined. If the left one is the real nearest, forward recreation is adopted, and the update operations will be redone. Otherwise, backward recreation is adopted, and the update operations will be undone.

V. EXPERIMENTAL EVALUATION

We mainly concentrate on the performance improvement in serving historical queries and the reduction in storage consumption provided by our proposed snapshot strategy. The average number of redone and undone operations in serving each historical query is taken as the criterion for evaluating the recreation performance, while the total number of snapshots is taken as the criterion for storage consumption.

A. EXPERIMENTAL SETUP

We adopt the Monte Carlo simulation method for evaluation. The three snapshot strategies (including the time interval based strategy, the update operation based strategy and our clustering method based strategy) are implemented in Python 3.7 with SKlearn. The computer is equipped with 8-core Intel i5 processor and 8GB of RAM.

Two kinds of datasets are required for evaluation. The first one is a sequence of arrival time instants of graph update operations, and the second one is a sequence of timestamps.
representing the historical states requested by the historical queries.

We use the dataset ‘CollegeMsg temporal network’ which is released by SNAP (Standford Network Analysis Project) at the website [32] for evaluation. The ‘CollegeMsg temporal network’ is a dynamic graph which had been updated 59,835 times spanning 193 days. The time instants of all the updates are available for generating the sequence of time instants of graph update operations. The statistics of the dataset are listed in TABLE 1.

In many applications, a minor part of the data tends to be frequently accessed while the majority part is rarely accessed. The Zipf distribution is widely adopted to describe the skewness of popularity in many real-world applications [33]. We also use the Zipf distribution to randomly generate the timestamps of the historical states requested by the historical queries.

B. EXPERIMENTAL RESULTS

1) ANALYSIS OF IMPROVEMENT IN RECREATION PERFORMANCE

In this experiment, the sequence of arrival time instants of graph update operations is composed of 59,835 elements as depicted in the above subsection. We totally generate 10,000 random values as the timestamps of the requested historical states. The frequency of the historical state being requested obeys the Zipf distribution with the parameter $\alpha = 1.2$. The number of snapshots $N$ varies from 10 to 200.

For each given number of snapshots, we use the three snapshot strategies to calculate the timestamps of the snapshots. After that, the average number of redone and undone operations in serving a historical query is obtained. We compare the three strategies in FIGURE 3. As shown, our proposed clustering based snapshot strategy greatly improves the recreation performance compared with the two traditional strategies.

As the number of snapshots increases, the average number of redone and undone operations decreases with all the three strategies. However, the storage consumption becomes higher as well. In practice, we should tradeoff the recreation performance and the storage consumption. As shown in FIGURE 3, the average number of redone and undone operations decreases sharply when the number of snapshots varies between 10 and 100, but very slowly between 100 and 200. Therefore, we choose 100 as the number of snapshots for evaluation.

For $N = 100$, we demonstrate the distribution of the number of redone and undone operations in serving each historical query in FIGURE 4. Each point $(x, y)$ in the curve means that the proportion of the historical queries that require no more than $x$ redone or undone operations equals $y$. As shown in the figure, the historical queries that require no more than 50 redone or undone operations nearly take a proportion of 77.2% with our cluster based snapshot strategy, while the proportion is 45.7% with the time-based strategy and only 8.7% with the operation-based strategy. The historical queries that require no more than 200 redone or undone operations take a proportion of nearly 100% with our cluster based snapshot strategy, while the proportion is about 90.3% and 58.2% respectively with the other two strategies.

We assign the Zipf distribution parameter $\alpha$ with 1.5, and keep all the other parameters unchanged. The results are shown in FIGURE 5, indicating that the superiority of our strategy is more apparent when the distribution of the historical queries is more skewed. As a comparison, we list the average number of redone and undone operations for each historical query with $N = 100$ in TABLE 2. The reduction of our strategy is between 70.7% and 95.5% compared with the two traditional strategies.

2) ANALYSIS OF REDUCTION IN STORAGE CONSUMPTION

We try to find the minimum number of snapshots to meet the recreation performance satisfying that the average number of redone and undone operations does not exceeding 15 and 100 respectively. The sequence of arrival time instants of graph update operations is also composed of 59,835 elements,

### TABLE 1. Statistics of CollegeMsg temporal network.

| Item                      | Value  |
|---------------------------|--------|
| Number of vertices        | 1,899  |
| Number of temporal edges  | 59,835 |
| Edges in static graph     | 20,296 |
| Time span                 | 193 days |
and we totally generate 10,000 random values as the timestamps of the requested historical states with the frequency of the historical state being requested obeying the Zipf distribution with the parameter $\alpha = 1.5$.

The comparison results are shown in FIGURE 6. With the same recreation performance guarantee, our strategy sharply reduces the storage consumption (nearly 78.9% on average). It should be noted that we do not differentiate the redoing or undoing cost of different kinds of update operations, and neither differentiate the storage cost of different snapshots.

In a word, our proposed strategy leads to recreation performance improvement and storage consumption reduction compared with traditional time-based and operation-based snapshot strategies. However, it should continually analyze the distribution of the historical queries and make adjustment dynamically, while the traditional ones require little analysis and make no dynamic adjustment. The analysis and adjustment process itself leads to computation overhead.

VI. CONCLUSION & FUTURE WORK

This paper studies the snapshot strategy for the ‘snapshot plus log’ solution to support historical queries of dynamic graphs. It points out that the inefficiency of the traditional snapshot strategies lies in the contradiction between the uniform distribution of snapshots and the skewed distribution of historical queries. In other words, the historical states are not equally requested, while the snapshots are evenly distributed under the traditional snapshot strategies. Therefore, this paper proposes a new snapshot strategy that determines the timestamps of the snapshots based on the distribution of the historical queries, such that the snapshots are near to the requested historical states with a very high probability. The experimental results show that the proposed strategy greatly improves the performance of historical state recreation and sharply reduces the storage consumption.

In the future, we will investigate how to apply this ‘snapshot plus log’ model in a distributed storage system of dynamic graph data. In a distributed environment, not only the snapshot strategy but also the placement strategy will affect the recreation performance. It is a big challenge to place the snapshots with variable timestamps onto distinct storage nodes to balance the historical query workload as well as to maximize the recreation performance.

REFERENCES

[1] R. A. Rossi, B. Gallagher, J. Neville, and K. Henderson, “Modeling dynamic behavior in large evolving graphs,” in Proc. 5th ACM Int. Conf. Web Search Data Mining (WSDM), New York, NY, USA, Feb. 2013, pp. 667–676.

[2] F. S. F. Pereira, S. D. Amo, and J. Gama, “Evolving centralities in temporal graphs: A Twitter network analysis,” in Proc. 17th IEEE Int. Conf. Mobile Data Manage. (MDM), Porto, Portugal, Jun. 2016, pp. 43–48.

[3] L. Yang, Z. Li, and A. Giua, “Containment of rumor spread in complex social networks,” Inf. Sci., vol. 506, pp. 113–130, Jan. 2020.

[4] C. Glacet, M. Fiore, and M. Gramaglia, “Temporal connectivity of vehicular networks: The power of store-carry-and-forward,” in Proc. IEEE Veh. Netw. Conf. (VNC), Kyoto, Japan, Dec. 2015, pp. 52–59.

[5] C. Küçükkeçeci and A. Yazıcı, “Big data model simulation on a graph database for surveillance in wireless multimedia sensor networks,” Big Data Res., vol. 11, pp. 33–43, Mar. 2018.

[6] M. Jalili, A. Salehzadeh-Yazdi, S. Gupta, O. Wolkenhauer, M. Yaghmaie, O. Resendis-Antonio, and K. Alimoghaddam, “Evolution of centrality measurements for the detection of essential proteins in biological networks,” Frontiers Physiol., vol. 7, p. 375, Aug. 2016.

[7] J. H. Davis and J. R. Cogdell, “Calibration program for the 16-foot antenna,” Elect. Eng. Res. Lab., Univ. Texas, Austin, TX, USA, Tech. Rep. NGL-006-69-3, Nov. 1987.

[8] W. Wan, F. Huang, X. Wang, Y. Zhang, and C. Zhang, “Dynamic detection of academic team communities based on temporal coauthor network,” in Proc. IEEE Int. Symp. Parallel Distrib. Process. Appl. IEEE Int. Conf. Ubiquitous Comput. Commun. (ISPA/IUCC), Guangzhou, China, Dec. 2017, pp. 464–469.

[9] Y. Sun, R. Barber, M. Gupta, C. C. Aggarwal, and J. Han, “Co-author relationship prediction in heterogeneous bibliographic networks,” in Proc. 17th IEEE Int. Conf. Adv. Social Netw. Anal. Mining, Kaohsiung, Taiwan, Jul. 2011, pp. 121–128.

[10] T. Tian, Y. Ren, Y. Feng, M. Zhou, H. Zhang, and J. Tan, “Modeling and planning for dual-objective selective disassembly using and or graph and discrete artificial bee colony,” IEEE Trans. Ind. Informat., vol. 15, no. 4, pp. 2456–2468, Apr. 2019.

[11] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, “Chronos: A graph engine for temporal graph analysis,” in Proc. 9th Eur. Conf. Comput. Syst., Amsterdam, The Netherlands, Apr. 2014, pp. 1–14.

| Strategies     | Redone/Undone operations ($\alpha=1.2$) | Redone/Undone operations ($\alpha=1.5$) |
|----------------|---------------------------------------|---------------------------------------|
| Time-based     | 113.8                                 | 71.0                                  |
| Operation-based| 155.4                                 | 160.9                                 |
| Cluster-based  | 33.4                                  | 8.0                                   |

FIGURE 5. Comparison of recreation performance with $\alpha = 1.5$.

FIGURE 6. Comparison of storage consumption with the same performance guarantee.
[12] U. Khurana and A. Deshpande, “Efficient snapshot retrieval over historical graph data,” in Proc. IEEE 29th Int. Conf. Data Eng. (ICDE), Brisbane, QLD, Australia, Apr. 2013, pp. 997–1008.

[13] J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graphs over time: Densification laws, shrinking diameters and possible explanations,” in Proc. Int. Conf. Knowl. Discovery Data Mining, Chicago, IL, USA, Aug. 2005, pp. 177–187.

[14] K. Semertzidis and E. Pitoura, “Durable graph pattern queries on historical graphs,” in Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE), Helsink, Finland, May 2016, pp. 541–552.

[15] G. Kolonari, D. Souravvias, and E. Pitoura, “On graph deltas for historical queries,” in Proc. Int. Workshop Online Social Syst., Istanbul, Turkey, Feb. 2013, pp. 1–6.

[16] G. Kolonari and K. Stefanidis, “Social search queries in time,” in Proc. 7th Workshop Personalized Access, Profile Manage., Context Awareness Databases, Riva del Garda, Italy, Jan. 2013, pp. 1–4.

[17] A. Kosmatopoulos, A. Gounaris, and K. Tsichlas, “Hinode: Implementing a vertex-centric modelling approach to maintaining historical graph data,” Computing, vol. 101, no. 12, pp. 1885–1908, Dec. 2019.

[18] L. Chen, Z. Liu, and W. Jia, “A snapshot system based on cloud storage log-structured block system,” in Proc. Int. Conf. Netw. Sens. Control, Miami, FL, USA, Apr. 2014, pp. 392–398.

[19] A. Zaki, M. Attia, D. Hegazy, and S. Amin, “Efficient distributed dynamic graph system,” in Proc. IEEE 7th Int. Conf. Intell. Comput. Inf. Syst. (ICICIS), Cairo, Egypt, Dec. 2015, pp. 465–471.

[20] A. G. Labouseur, J. Birnbaum, P. W. Olsen, S. R. Spillane, J. Vijayan, J.-H. Hwang, and W.-S. Han, “The G∗ graph database: Efficiently managing large distributed dynamic graphs,” Distrub Parallel Databases, vol. 33, no. 4, pp. 479–514, Dec. 2015.

[21] B. Yu, “Research on storage system of large scale dynamic evolving graph,” M.S. thesis, School Comput. Sci. Technol., Xidian Univ., Xi’an, China, 2017.

[22] W. Han, K. Li, S. Chen, and W. Chen, “Aaxo: A temporal graph management system,” Big Data Mining Anal., vol. 2, no. 1, pp. 58–71, Mar. 2019.

[23] S. Aridhi, A. Montresor, and Y. Velegrakis, “BLADYG: A graph processing framework for large dynamic graphs,” Big Data Res., vol. 9, pp. 9–17, Sep. 2017.

[24] L. Xiangyu et al., “Efficient Snapshot Strategy for Dynamic Graph Storage Systems to Support Historical Queries,” Adv. Mater. Res., vol. 8, no. 1, pp. 1346–1357, 2015.

[25] M. Nelson, S. Radhakrishnan, and C. N. Sekharan, “Algorithms on compressed time-evolving graphs,” in Proc. IEEE Int. Conf. Big Data (Big Data), Los Angeles, CA, USA, Dec. 2019, pp. 227–232.

[26] Y. Yang, W. Y. Li, M. N. Zhang, X. M. Ding, and J. Dai, “Using Zipf distribution to predict popularity data for storage systems,” Adv. Mater. Res., vol. 8, no. 1, pp. 985–994, 2014.

[27] S. Aridhi, A. Montresor, and Y. Velegrakis, “Finding lasting dense subgraphs,” Data Mining Knowl. Discovery, vol. 33, no. 5, pp. 1417–1445, Sep. 2019.

[28] Y. Yang, W. Y. Li, M. N. Zhang, X. M. Ding, and J. Dai, “Using Zipf distribution to predict popularity data for storage systems,” Adv. Mater. Res., vol. 985–994, pp. 2152–2155, Jul. 2014.

[29] A. Zaki, M. Attia, D. Hegazy, and S. Amin, “Comprehensive survey on no. 4, pp. 479–514, Dec. 2015.

[30] A. Zaki, M. Attia, D. Hegazy, and S. Amin, “Comprehensive survey on no. 4, pp. 479–514, Dec. 2015.

[31] A. Zaki, M. Attia, D. Hegazy, and S. Amin, “Comprehensive survey on no. 4, pp. 479–514, Dec. 2015.

[32] A. Zaki, M. Attia, D. Hegazy, and S. Amin, “Comprehensive survey on no. 4, pp. 479–514, Dec. 2015.