LSM-Subgraph: Log-Structured Merge-Subgraph for Temporal Graph Processing

Jingyuan Ma[1], Zhan Shi[1(✉)], Shang Liu[2], Wang Zhang[1], Yutong Wu[1], Fang Wang[1], and Dan Feng[1]

[1] Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
{jyma,zshi,zhangtw,yutongwu,wangfang,dfeng}@hust.edu.cn
[2] Graduate School of Informatics, Kyoto University, Kyoto, Japan
liu.shang.33s@st.kyoto-u.ac.jp

Abstract. Temporal graphs, with a time dimension, are attracting increasing interest from research communities. Existing temporal graph storage formats mainly include copy-based models, log-based models, and hybrid models that have emerged in recent years. Neither the copy-based model nor the log-based model can trade-off storage and query time well. Hybrid models try to find a compromise between the above two models, but existing models do not consider the skewness of vertex degree in temporal graphs is changing over time. Based on these considerations, we propose LSM-Subgraph, a hybrid storage format that only stores snapshots divided by the fluctuation-aware method and in-between logs. First, LSM-Subgraph uses a PMA-based snapshot creation model to store snapshots based on packed memory arrays (PMA), avoiding rebuilding the whole data structure. Second, LSM-Subgraph uses a select-timepoint method based on fluctuation-aware to divide shards during the update, which achieves a good tradeoff between storage overhead and query time cost. Extensive experimental evaluations over various real-world graphs illustrate that LSM-Subgraph outperforms state-of-the-art temporal graph systems in both memory and time consumption.

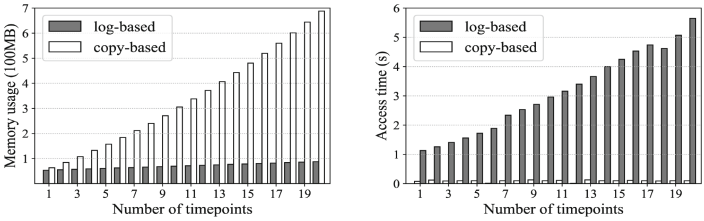
Keywords: Temporal graph · Packed-memory array · Graph storage · Graph analysis

1 Introduction

Graph, as a fundamental data model, has been widely used in many domains [9,13,21]. Real-world graphs are not only large in size but also often evolve over time as the connections and relationships change. For example, Facebook [23] social network has more than one billion nodes. Among them, 86,000 objects are changing and 2.5 emails are sending out per second on average. These time-evolving graphs are called temporal graphs [5], whose vertex/edge insertion and deletion can happen over time.

B. Li et al. (Eds.): APWeb-WAIM 2022, LNCS 13421, pp. 477–494, 2023.
https://doi.org/10.1007/978-3-031-25158-0_39

Temporal graph processing is challenging in how to trade-off between storage and query time. This is because as a graph evolves constantly, it produces a large number of snapshots as time flies. Each graph snapshot in history should be available. Storing all the snapshots consume memory resource excessively, while disk storage is slow for graph accessing and processing. Besides, how to switch snapshot quickly is not trivial.



(a) Storage cost (b) Access time

Fig. 1. Overhead comparison

Modern out-of-core graph processing systems store graphs in persistent storage [10]. By loading graphs at desired time points into memory and replacing the previously processed graphs, they finish temporal graph processing. Such graph loading is costly for temporal graph processing. Existing temporal models to handle this problem fall into two types: copy-based models [3,7,12,15] and log-based models [4,8].

Copy-Based Model: It consists of a sequence of snapshots, e.g. *FVF* [15]. Each of which represents a specific state at a single time point. This model generates and stores the copies of the graph at different time points. It stores the graph structure completely. Hence it favors structure locality well and does well in queries. However, there is high redundancy in consecutive snapshots storage, which is inefficient in memory for large-scale dynamic graphs processing.

Log-Based Model: It is composed of a series of events where can be insertions or deletions of nodes and edges, e.g. DeltaGraph [8]. Log-based model stores simple updates at each time point. This model stores only incremental updates of neighbouring snapshots, thus avoiding the storage redundancy of copy-based model. However, it introduces extra overhead at query time for reconstructing snapshots as of specified time points.

We examine the performance of the copy-based model and the log-based model as shown in Fig. 1. We evaluate these two models using the Wiki-talk dataset [11], which is the temporal graph of users editing talk pages on Wikipedia. Each directed edge has a timestamp t. The first 40% is treated as the initial snapshot (e.g. Snapshot 0), and the rest is divided into 20 time-points' updates evenly. The experimental results show that the copy-based model incurs up to 7× memory cost of the log-based model (Fig. 1(a)), while the access time

of the log-based model is up to 5× more than that of the copy-based model (Fig. 1(b)). There is an obvious tradeoff between memory and time consumption. We need a new temporal model to support efficient queries with lower storage overhead.

To solve the above problems, hybrid models have been proposed, such as Pensieve [22]. Based on the skewness of vertex degree, Pensieve combines the copy and log model. The log model is used for high-degree vertex data, and the copy model is used for low-degree vertex data. However, the skewness in real-world graphs usually changes over time. In Fig. 2, we examine the Hits@100 of the real-world temporal graph datasets from SNAP [11]. Hits@100 is the proportion of 100 high-degree vertices in the initial snapshot that are still high-degree vertices after being updated. Figure 2 shows that some vertices will no longer be high-degree vertices over time. The degree of a vertex may either grow or shrink in the future, sometimes changing drastically, which is not concerned in Pensive. As a result, Pensieve's design may cause vertex's degree unfit for the model over time and affect the system's performance ultimately.

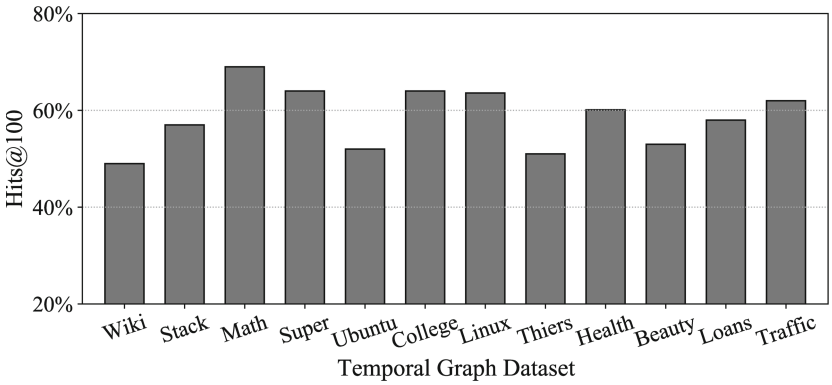


Fig. 2. Hits@100 with real-world temporal graph dataset

The skewness of vertex degree is considered in many systems, but they are based on this static characteristic to design corresponding data structures. However, the skewness of vertex degree in temporal graphs is changing over time, and the data structure designed for the degree skewness is not applicable. Therefore, the main challenge is how to design a data structure according to the changing skewness of vertex degree and how to choose the time point for inserting the key snapshot.

In this work, we propose a temporal graph storage model based on log-structured merge-subgraphs, called LSM-Subgraph. Three factors contribute to the efficiency of LSM-Subgraph. First, it splits all logs into multiple data shards which are composed of a log array and an adjacency array. Specifically, the log array keeps the updates in the edge list format and is designed to accelerate

data ingestion. At the same time, the adjacency array based on Packed Memory Arrays [1,2] stores a snapshot of the initial moment which is to solve the problem of snapshot reconstruction caused by the update. Second, by using a fluctuation-aware method, LSM-Subgraph sets a snapshot according to the size change during the evolution of the graph. The next snapshot will be created when the relative change rate of graph size reaches the threshold. Third, LSM-Subgraph reduces the size of logs during the query by utilizing a log-merging method. The edge list format logs will firstly merge in memory and then apply to the adjacency array, which can avoid repeated operations.

We implement our system on top of Ligra [16], a famous graph processing system. For comparison, we evaluate it against two traditional temporal graph processing systems: GraphPool and Chronos, and a hybrid model temporal graph system: Pensieve. The experiment results show that LSM-Subgraph supports faster queries with lower storage overhead than existing systems.

2 Background and Related Work

2.1 Graph Storage Formats

There are three fundamental graph storage formats: CSR (Compressed Sparse Row), adjacency list, and adjacency matrix.

CSR is a popular format for storing sparse graphs [14]. It uses a node array and an edge array to store graph structure. The node array records starting index information of every vertex, and destination vertices are stored in the edge array. The obvious advantages of CSR are compactness and access efficiency. But a simple change of a node or an edge will lead to rebuilding the whole node or edge array. For example, when a new edge is inserted, all elements after it need to slide one by one.

Adjacency list is also a sparse graph format [9]. It keeps an array of nodes and each node stores a pointer to a linked list of edges. Adjacency list supports insertions or deletions well. For example, when a new edge needs to be inserted, we just update the corresponding edge list instead of changing the whole graph. Though it supports fast insertions, the search efficiency is not high because each element is unsorted.

Adjacency matrix stores an $n \times n$ matrix for a graph with n nodes. It excels in storing dense graphs. However, real-world graphs are generally sparse, so it wastes more storage than both CSR and adjacency list. To make it worse, any update of elements will bring the reconstruction of the whole graph. Hence, adjacency matrix is not suitable for dynamic graphs.

2.2 Packed Memory Arrays

Packed Memory Arrays (PMA), a sparse-array structure, has been widely used in many applications [1,6,17,19]. For instance, APMA [1] gives the first adaptive packed-memory array that adjusts to the input pattern automatically. It

performs better than traditional PMA, and achieves only $O(logN)$ amortized element moves. RMA [6] proposes an improved version of sparse arrays, Rewired Memory Array, which fixes its major flaws and adds a new technique of memory rewiring into its rebalancing mechanism.

PMA stores N elements in a sorted array, interleaved with empty spaces or gaps. These gaps serve the purpose of providing extra space to fast insert new items in an arbitrary position of the array, instead of moving existing elements. The insertion, deletion, and rebalance operations are as follows.

To insert a new element, we have to find its target position using a binary search. If the cell has been occupied by the other element e, we need to move e towards the nearest empty gaps. To delete an arbitrary element, we just mark this cell as NIL. When element insertion or deletion causes the data structure to be locally dense or sparse, PMA adjusts the position of empty elements in the data structure to meet the local threshold setting.

To examine the performance of PMA, we compare it with traditional data structure CSR. From experimental results, when 70 million data is inserted, the update time of CSR is 40 s while that of PMA is only 2 s. Therefore, PMA is beneficial to design the temporal graph storage format.

However, unlike classical dynamic datasets, the temporal graph is not just skewed, its skewness also changes with time, which makes it hard to determine the number and distribution of empty elements in PMA, and the traditional rebalancing operations are inapplicable. Thus PMA cannot be applied directly. In response to these challenges, we design a PMA-based adjacency array storage model.

2.3 Temporal Graph Storage

There is a growing interest in analyzing and computing temporal graphs. Every temporal system designs and improves temporal graph storage formats for their own applications. For example, GraphOne [9] stores temporal graphs using copy+log approach. It is a unified graph data store abstraction that offers data access at different granularity for various real-time analytics and queries, while supporting high arrival velocity of fined-grained updates simultaneously. But the updated data is stored in adjacency list periodically, losing the temporal property. Pensieve [22] proposes a skewness-aware multi-version graph processing system. It uses a differentiated graph storage strategy that stores high degree vertices using log-based scheme, while stores low degree vertices using copy-based scheme. However, temporal graph is changing over time, and the degree of vertices is not predicted in the future. Huanhuan Wu [20] proposes an equal-weight damped time window model, to tradeoff between the required storage space and the information loss. It is clear that this system achieves excellent storage compactness with the sacrifice of accuracy. DeltaGraph [8] is a distributed hierarchical structure that enables compact storage of the historical trace of a network. It uses the log-based scheme to change the full graph structure to partial storage for modification. This system reduces storage overhead, but suffers

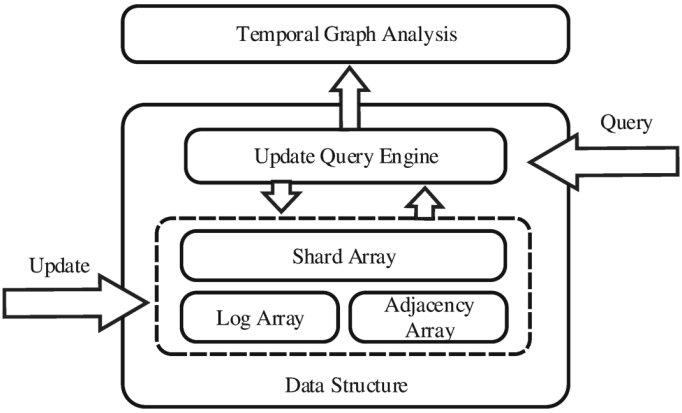


Fig. 3. Architecture of LSM-Subgraph

longer construction time. SAMS [18] presents a novel approach to execute algorithms concurrently for multiple graph snapshots, which is a copy-based model. By leveraging discrepancies between the analyzed graph snapshots, it accelerates temporal analysis. But it just focuses on multiple snapshots' analysis, not on temporal graph storage.

In a word, existing methods fail to achieve a good tradeoff between storage cost and query time overhead during temporal graph processing. None of them consider the characteristic that the skewness of vertex degree is changing over time, which is nonnegligible for improving the performance of temporal graph processing.

3 LSM-Subgraph Design

In this section, we begin with an overview of LSM-Subgraph architecture. Then we present details of the system design.

3.1 Overview

LSM-Subgraph uses a hybrid graph storage format that consists of key snapshots and in-between logs. Specially, we select a few key time points to store snapshots using the copy-based model and store in-between updates in the log-based model. There are two main challenges to deal with: 1) How to create a designated snapshot with the key snapshot and logs instead of reconstructing; 2) How to select the key time points to store graphs in the copy-based model.

To handle them, we elaborate a PMA-based adjacency array model to store snapshots. Then, there is a crucial tradeoff between storage cost and query time overhead when selecting key time points. We design a fluctuation-aware snapshot creation method to determine which format to store graphs at the designated time point (discussed in Sect. 3.3). Finally, We propose a method to reduce the number of log data merged in the query process.

Figure 3 depicts an overview of LSM-Subgraph architecture. LSM-Subgraph mainly includes LSM-Subgraph data structure and update query engine. LSM-Subgraph splits all logs into multiple data shards, and each shard has a log array and an adjacency array based on Packed-Memory Array (PMA). The adjacency array based on PMA handles the reconstruction problem by reserving some space for updating. The function of the update query engine is to accomplish some operations during the update and query. For the coming updated edges, it first computes the discrepancy between the last snapshot and the temporary one currently (defined as TD). If TD goes above the threshold, the temporary snapshot is stored in the copy-based model, and corresponding logs are stored in the log-based model. In the query phase, LSM-Subgraph computes and finds the latest time point first. And then it loads the key snapshot and corresponding logs. After merging logs into the key snapshot, LSM-Subgraph constructs the target snapshot for temporal graph analysis.

3.2 PMA-Based Adjacency Array Model

The PMA-based adjacency array consists of a vertex array and the corresponding adjacency edge array, as shown in Fig. 4. The maximum number of vertices in the current snapshot ID_{max} determines the size of the vertex array because the number of vertices in real-world datasets is much smaller than the number of edges. The vertex array stores all vertex IDs and the degree of each vertex, that is, $0 \sim ID_{max}$. If the vertex does not exist, the corresponding degree is assigned a value of 0. When the vertex degree Deg is greater than 0, the vertex pointer points to the corresponding adjacent edge array. All target vertices in the adjacent edge array are arranged according to the size of ID. In order to ensure the efficient insertion and deletion of updated data, certain empty gaps are reserved for the edge array elements to avoid the elements moving backward as a whole to reduce update overhead. The PMA-based adjacency array is introduced in detail below.

The Size of the Reserved Space: According to the size of memory space and requirements of the graph processing task, we specify the total number of empty elements and then assign empty elements to each vertex according to the degree of each vertex. The calculation process is as Eq. 1:

$$Gap_i = \frac{Gap_i}{Deg_{sum}} * Deg_i \tag{1}$$

Among them, Gap_i is the number of empty gaps allocated to the i vertex adjacency array, Gap_{sum} is the sum of empty gaps, Deg_{sum} is the sum of the degrees of all vertices. If there are no special instructions, in this experiment, Gap_{sum} is set to Deg_{sum}, that is, the number of empty gaps in the adjacent array of each vertex is equal to the size of the vertex's out-degree.

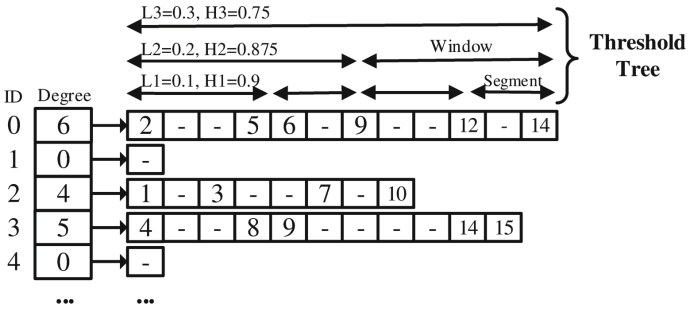


Fig. 4. Sample layout of the key snapshot

The Distribution of Empty Gaps: The traditional methods of determining the distribution of empty gaps TPMA and APMA do not consider the time-evolving feature of temporal graphs, and this paper proposes a new empty gaps distribution method, as shown in Fig. 4. The number of empty gaps between adjacent elements is determined by the difference between the vertex numbers and the total number of empty gaps, as shown in the Eq. 2:

$$Num_{gap} = \frac{Gap_i}{ID_{max} - ID_{min}} * (ID_2 - ID_1) \tag{2}$$

where Gap_i is the total number of empty gaps in the adjacent array of vertices i, ID_{max} and ID_{min} are the maximum and minimum non-empty elements of the existing element numbers in the edge array respectively. ID_1 and ID_2 are the numbers of adjacent non-empty elements before and after the target insertion position. The main reason for using this method is that when the updated log of the temporal graph merges into the initial snapshot, the state of an edge that has been updated several times in the period only saves the latest state. Therefore, taking into account the irregularity of the temporal graph update, LSM-Subgraph reserves appropriate storage space for all possible adjacent edges.

Rebalance: When the update logs apply to the snapshot, the local density in the adjacent array could be too high or too low, and the entire snapshot needs to rebalance. The local density of a specified area is defined in Eq. 3:

$$Density = \frac{Num_{element}}{Num_{element} + Num_{gap}} \tag{3}$$

Among them, $Num_{element}$ and Num_{gap} represent the number of non-empty elements and empty gaps in the local area, respectively. To better apply to the update of temporal graphs, LSM-Subgraph has improved the existing rebalancing operations. As shown in Fig. 4, LSM-Subgraph constructs a 3-level calibration tree for the adjacency array of each vertex. Each layer has a corresponding threshold. When the updated edge is applied to the snapshot and causes the local density to be too high or too low, the rebalancing operation will be triggered.

Two adjacent blocks will merge to meet the threshold requirement. The worst case is that too many insert operations cause the entire array can not meet the threshold requirements. The array needs to copy to new storage space.

When querying other snapshots at arbitrary time points, LSM-Subgraph first finds the latest key time point. After loading the key snapshot and corresponding logs, it displays the updates on the key snapshot using logs. This process of switching can be represented by Formula 4,

Algorithm 1. LSM-Subgraph construction

Input: Set of updated edges U
Output: Set of key snapshots S, Set of in-between logs L

```
for edge e in U do
    if snapshot_i.NotfindNode(e.src) then
        snapshot_i.addNode(e.src);
    end if
    snapshot_i.addEdge(e);
    log_i.addEdge(e);
    if computeDiscrepancy(snapshot_i,snapshot_{i-1})> β then
        S.addSnapshot(snapshot_i);
        L.addLog(log_i);
    else
        continue;
    end if
end for
return S, L
```

Algorithm 2. LSM-Subgraph query

Input: Set of key snapshots S, Set of in-between logs L, Target time point i
Output: $snapshot_i$

```
for timepoint t in L do
    if |t − i|.isMinimum() then
        break;
    end if
end for
if t < i then
    snapshot_t.addEdge(L, t, i);
else
    snapshot_t.deleteEdge(L, t, i);
end if
return snapshot_i
```

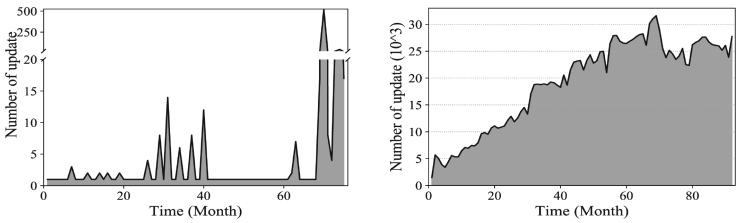
$$S_k + L_{ki} \rightarrow S_i, \tag{4}$$

where we use the key snapshot S_k and corresponding L_{ki} to generate the target snapshot S_i.

Algorithm 1 shows the process of snapshot construction. For each edge in the updated set, LSM-Subgraph first constructs a temporary snapshot. When discrepancy exceeds threshold, LSM-Subgraph stores a key snapshot with corresponding logs. Algorithm 2 presents the procedure of query. LSM-Subgraph first loads the latest key snapshot and corresponding logs. Then it merges them into a target snapshot.

3.3 Fluctuation-Aware Snapshot Creation Method

Besides the PMA-based adjacency array model, it is also important to decide when to create snapshots. There is a crucial tradeoff on selecting the interval between snapshots. If the interval is too large, the log will be too large to query effectively. On the contrary, there will be more redundancy between adjacent snapshots, which wastes space.



(a) Wikitalk (b) Stackoverflow

Fig. 5. The updated characteristic of temporal graphs

We have two alternative methods to solve this problem. One obvious method is to select key snapshots based on time period, for example, the next key snapshot is created every ten minutes or one hour. However, we have to consider a terrible situation where there are many update operations in the current interval, but few or no update operations in the next one. As shown in Fig. 5, every real-world temporal graph owns itself characteristic over time and this changing characteristic can not be concluded or predicted. The imbalance of distribution will increase the overhead of storage and query. The other is to set up key snapshots based on the size of logs, that's to say, the next snapshot is created when the size of logs reaches the threshold. This scheme can avoid the imbalance of the method based on period. But both insertions and deletions of the same nodes or edges may be in the same period. The next key snapshot would still be created even if the absolute size of logs is less than the threshold, where the absolute size is the number of edges when the system merges all additions and deletions of the same elements. It will result in structural redundancy between adjacent snapshots.

Inspired by FVF [15], we design a fluctuation-aware snapshot creation method to select key time points based on the above discussions. The degree of difference between adjacent snapshots can be defined as:

$$TD(K_1, K_2) = \frac{|E_G|}{|E_{K_1}| + |E_{K_2}|} \tag{5}$$

Among them, $|E_G|$ represents the number of all edges in graph G. $|E_{K_1}|$ and $|E_{K_2}|$ are the number of all edges in the two snapshots. When LSM-Subgraph stores updated data, it first maintains a temporary snapshot K_t in the system. This snapshot adopts the PMA-based adjacency array storage model and then calculates the temporary snapshot and the initial snapshot K_l in the latest data shard in LSM-Subgraph. If $TD(K_l, K_t) > \beta$, where β is a user-defined threshold, create a new data shard and save the temporary snapshot as the initial snapshot, otherwise continue to store the log and update the temporary snapshot.

In order to guarantee the query efficiency and space-saving, we have to find the optimal threshold β, for it affects the tradeoff between storage overhead and access time. We compare the performance over various temporal graphs with different parameters(present in Sect. 4.4). We analyze how the memory usage and access time change with the threshold β in real-world graph dataset. Let M_k represents the average memory overhead of data shards, N represents the number of data shards and M_l represents the memory overhead of the log array. The memory overhead of LSM-Subgraph is $Memory = N * M_k + M_l$. The log array in LSM-Subgraph stores all update edge data, so $M_l = O(E)$, for the same temporal dataset, M_l can be regarded as a constant. As the threshold β increases, N continues to decrease, M_k continues to increase, and finally $Memory$ continues to decrease(see Sect. 4.4 for detail), indicating that N has a much

Algorithm 3. Log Merge

Input: Set of edges list to be updated U
Output: Set of edges list to be updated U'

```
1: U.sort();
2: flag1=U.first();flag2=U.second();
3: while flag1<=U.end()&&flag2<=U.end() do
4:    if flag1 == flag2 then
5:       flag1.delection();
6:       flag1 = flag2;
7:    else if flag1 == -flag2 then
8:       flag1.delection;
9:       flag1 = flag2.next();
10:      flag2.delection;
11:   else
12:      flag1 = flag2;
13:   end if
14:   flag2 = flag1.next();
15: end while
16: return U'
```

greater impact on the memory overhead than M_k, so we can regard M_k as a constant, and $N \propto \frac{1}{\beta}$. As the threshold β increases, the memory overhead of LSM-Subgraph will continue to decrease. The relationship between $Memory$ and β can be defined as:

$$Memory = M_k * f(\frac{1}{\beta}) + M_l \tag{6}$$

For access time, it consists of the time T_k to find the corresponding data shard and the query time T_l within the data shard. The access time of LSM-Subgraph is $AccessTime = T_k + T_l$. As the PMA-based adjacency array is used to store the snapshot, the time to find the corresponding data shard is much less than average the query time within the data shard, so T_k can be regarded as a constant. As the threshold β increases, the memory overhead by a single snapshot increases. Therefore β and T_l are positively correlated. The relationship between $AccessTime$ and β can be defined as:

$$AccessTime = g(\beta) + T_k \tag{7}$$

According to the above analysis, we know access time and storage usage always intersect in one point. We find that setting the threshold at 0.03 is optimal in most cases(present in Sect. 4.4). Hence, we recommend the threshold β is 0.03. If not specified, we set the $\beta = 0.03$ in the following experiments.

3.4 Log-Merging Method

The query process needs to find the initial snapshot and log data according to the time point, and then merge to generate the snapshot data at the target

Table 1. Details of temporal graph datasets

Datasets	Nodes (M)	Edges (M)	Description
Youtube	1.13	2.99	static, Youtube users
Wikitalk	1.14	7.83	dynamic, Wikipedia users
Wiki-topcat	1.79	28.51	static, Wikipedia hyperlinks
Soc-pokec	1.63	30.62	static, Pokec connections
Stackoverflow	2.60	63.50	dynamic, Stackoverflow interactions
Livejournal	4.84	68.99	static, LiveJournal friendships
Wikipedia	2.1	66.9	static, Wikipedia links (fa)
Orkut	3.07	117.19	static, Orkut connections
Twitter	61.6	1470	static, Twitter connections

time. In the temporal graph update process in the real world, the same element (including vertices and edges) will be updated multiple times at different times. Therefore, we propose a log-merging method to further reduce the amount of log merged data. Assume that the update log within a period time is $Log = e_{1,1}, e_{2,2}, e_{3,3}, e_{3,4}, e_{3,5}, e_{4,6}, -e_{4,7}, ..., e_{n,t}$, where t is the update time, n is the number of the updated edge, e indicates the edge insertion operation, and $-e$ indicates the edge deletion operation. It can be seen that the edge e_3 is continuously and repeatedly inserted at the time $t = 3, 4, 5$, and the edge e_4 is inserted and deleted at the time $t = 6, 7$. Algorithm 3 presents the Log-Merging method. For the same update operation continuously, the snapshot only records the latest update status, that is, the update at time $t = 5$. For reciprocal update operations, the final update operation performed is 0. So the above update log can be expressed as $Log = e_{1,1}, e_{2,2}, e_{3,5}, ..., e_{n,t}$.

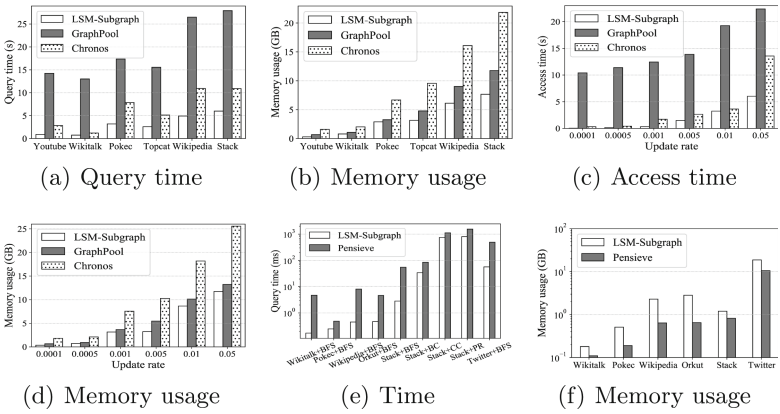
4 Evaluation

4.1 Experiment Setup and Datasets

We evaluate LSM-Subgraph on a commodity multi-core machine. It is equipped with a 16-core 1.60 GHz Intel(R) Xeon(R) CPU E5-2603, 128 GB of memory. The program is compiled with g++ version 11.0.

We use nine real-world graph datasets of different scales from SNAP [11], Twitter and Konect. The details of graphs describe in Table 1. Wikitalk and Stackoverflow are temporal graph datasets, and their periods are 2320 days and 2774 days, respectively. Stackoverflow is the largest temporal network that we can find on SNAP. To store all snapshots of copy-based models in the limited memory, we split the dataset evenly into updates of 60's time points based on days. In terms of static graph datasets, we generate random updates by adjusting the parameters including update rate, add rate. Update rate denotes the proportion of updated edges in the graph. Add rate indicates the proportion of additional edges in the update. The default update rate and add rate are 0.001 and 0.9, respectively.

We use the following algorithm as a benchmark to test the performance of the system: **2-hop:** 2-hop neighbour query, **BFS:** Breadth first search, **CC:** Find Connected Components, **BC:** Compute betweenness centrality of vertices, **PR:** Pageranks of vertices.



(a) Query time (b) Memory usage (c) Access time

(d) Memory usage (e) Time (f) Memory usage

Fig. 6. Overall performance comparison

4.2 Overall Performance Comparison

Compare with Traditional Models: We select temporal graph systems Chronos [3] and GraphPool [8] to compare against LSM-Subgraph. In terms of basic graph storage formats, Chorons is a copy-based model and GraphPool is a log-based model. We select the 2-hop neighbour query to examine the query performance, where the query time includes both snapshot construction time and 2-hop neighbour compute time.

First, we test datasets of different sizes of temporal graphs in the same update scenario, and the results show in Fig. 6(a) and 6(b). Compared with GraphPool, the query efficiency of LSM-Subgraph is improved by an average of 86%, and the memory overhead can be reduced by 9% to 57%. That is because GraphPool needs to build a dedicated snapshot for each designated moment when querying, and a bitmap is created for each update edge, which increases storage costs. The query efficiency of LSM-Subgraph is 53% higher than that of Chronos on average, and the memory overhead is much smaller than it. That is because Chronos needs to traverse all the bitmaps of the edge when querying and it stores data in the form of snapshots at every moment, resulting in spatial redundancy.

Second, for different update scenarios, we test the memory overhead and query time on the same dataset Wikitalk. The results are shown in Fig. 6(c) and 6(d). Compared with GraphPool, the query time cost of LSM-Subgraph is reduced by 90% on average. The main reason is that the PMA-based adjacency array model of LSM-Subgraph reserves a certain storage location. A small number of updates can be done directly using these empty locations, avoiding Data structure changes and even reconstruction. At the same time, memory overhead has been reduced by an average of 26%. Compared with Chronos, LSM-Subgraph's query time has been reduced by an average of 58%. Memory overhead has been reduced by 64% on average. The main reason is that LSM-Subgraph divides all logs into several data shards, and only one initial snapshot is kept in each data shard, thereby reducing memory overhead.

Compare with Hybrid Model: We compare the performance of LSM-Subgraph with Pensieve using six real-world datasets. Figure 6(e) compares the average querying time of different systems with different datasets and graph algorithms. The result shows that the average querying time of Pensieve is much larger than LSM-Subgraph, up to 12.5×. The reason is that the Pensieve'snapshot needs to be reconstructed for each query. The further away from the moment of the root node, the longer it will take to rebuild the snapshot. The query time of Pensieve at the time point far away from the root node is several times that of the near root node. The average memory overhead of LSM-Subgraph is 3.2× of Pensieve because it uses a fluctuation-aware method to store multiple snapshots. Moreover, the increase in memory is within a tolerable range and is far less than the cost of query reduction.

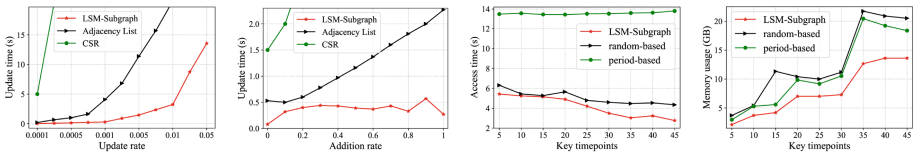


Fig. 7. Performance comparison within LSM-Subgraph

4.3 Comparison Within LSM-Subgraph

We examine the efficiency of each strategy in LSM-Subgraph, including PMA-based adjacency array model and fluctuation-aware snapshot creation method. As for PMA-based snapshot creation model, we examine the update efficiency of LSM-Subgraph against traditional graph storage formats. In terms of the fluctuation-aware snapshot creation method, we evaluate fluctuation-based against period-based and random-based. In this part, all experiments are conducted on temporal dataset Stackoverflow. When update rate varies, all addition rate is set to 0.9, and all update rate is fixed to 0.001 while addition rate changes.

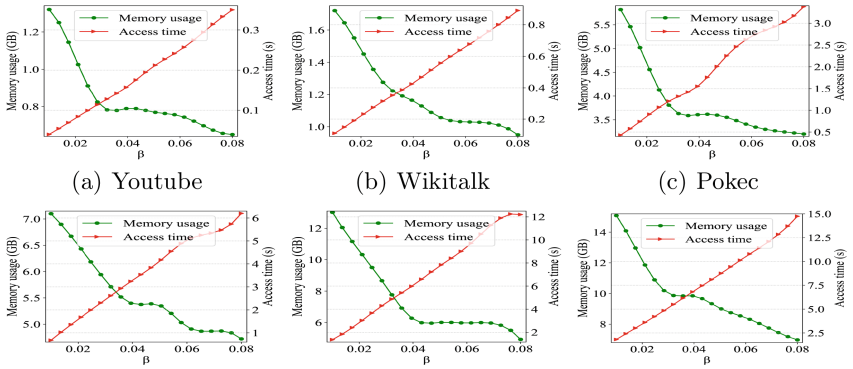
Figure 7(a) shows update performance under different update rates. We can find that the update time of three models increases with the growth of the update rate. When update rate is more than 0.0001, CSR takes unbearable time to rebuild graphs. And when update rate is over 0.0075, the reconstruction time of adjacency list is intolerable. When the number of updated edges is not rather large, LSM-Subgraph uses slots to complete these insertions or deletions. As the update rate grows, slots in segment may be not suitable for updates, incurring some rebalancing operations and increasing the update time.

We further evaluate the influence of addition rate, which represents the skewed distribution of insertions and deletions in updated edges. Figure 7(b) compares the update time on different addition rates. CSR and Adjacency list are sensitive to the number of additional edges. Particularly, we can not compute the update time of CSR when addition rate is more than 0.1. The update

time of adjacency list grows linearly as the addition rate increases. In contrast, LSM-Subgraph stays stable whatever the size of additions is.

Then we analyze the performance of key snapshot choice. Figure 7(c) depicts the access time of various key timepoints, where different timepoints mean the number of key snapshots in system. We can find that it costs the least time for LSM-Subgraph to access the graph at arbitrary time point against other strategies. And the period-based method costs up to 3× time of LSM-Subgraph and the number of key timepoints has little influence on it.

In Fig. 7(d), we compare the memory usage of different strategies. The memory usage of them increases as the number of timepoints grows, because system needs more space to store key snapshots and logs. Whatever the number of timepoints is, the storage overhead of LSM-Subgraph is less than both random-based strategy and period-based strategy. This is because LSM-Subgraph reduces the redundancy between neighbouring snapshots significantly.



(a) Youtube (b) Wikitalk (c) Pokec

(d) Topcat (e) LiveJournal (f) Stackoverflow

Fig. 8. Performance of different parameters over various graphs

4.4 System Design Parameters

We further verify and analyze the rationality of the selection of the difference threshold during the update process. Figure 8 shows the comparison of memory overhead and query time cost under different threshold parameters when LSM-Subgraph processes different real graph datasets. It can be observed from the figure that when β is approximately 0.03, LSM-Subgraph can achieve a good tradeoff between storage and query. Therefore, we set the fluctuation threshold as 0.03 in our experiments.

5 Conclusion

In this work, we observe that the skewness of vertex degree in temporal graphs is changing over time. Based on our findings, we propose LSM-Subgraph, a

temporal graph storage model based on log-structured merge-subgraph. LSM-Subgraph uses a PMA-based adjacency array model to store vertex data and leverages a Fluctuation-Aware snapshot creation method to create snapshots at key time points. Through experiments, we determined the threshold β, under which a comprehensive experiment was carried out to evaluate the performance of the design. Results show that LSM-Subgraph supports faster queries with lower storage overhead than existing systems.

References

1. Bender, M.A., Hu, H.: An adaptive packed-memory array. In: Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 20–29. PODS (2006)
2. De Leo, D., Boncz, P.: Packed memory arrays - rewired. In: 2019 IEEE 35th International Conference on Data Engineering, pp. 830–841 (2019)
3. Han, W., et al.: Chronos: a graph engine for temporal graph analysis. In: Proceedings of the Ninth European Conference on Computer Systems. EuroSys (2014)
4. Haubenschild, M., Then, M., Hong, S., Chafi, H.: Asgraph: a mutable multi-versioned graph container with high analytical performance. In: Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, pp. 1–6 (2016)
5. Holme, P., Saramäki, J.: Temporal networks. Phys. Rep. **519**(3), 97–125 (2012)
6. Itai, A., Konheim, A.G., Rodeh, M.: A sparse table implementation of priority queues. In: Proceedings of the 8th Colloquium on Automata, Languages and Programming, pp. 417–431 (1981)
7. Ju, X., Williams, D., Jamjoom, H., Shin, K.G.: Version traveler: fast and memory-efficient version switching in graph processing systems. In: 2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 2016), pp. 523–536 (2016)
8. Khurana, U., Deshpande, A.: Efficient snapshot retrieval over historical graph data (2013)
9. Kumar, P., Huang, H.H.: Graphone: a data store for real-time analytics on evolving graphs. ACM Trans. Storage (2020)
10. Kyrola, A., Blelloch, G., Guestrin, C.: Graphchi: large-scale graph computation on just a PC. In: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2012), pp. 31–46, October 2012
11. Leskovec, J., Krevl, A.: SNAP datasets: Stanford large network dataset collection, June 2014
12. Macko, P., Marathe, V.J., Margo, D.W., Seltzer, M.I.: Llama: efficient graph analytics using large multiversioned arrays. In: 2015 IEEE 31st International Conference on Data Engineering, pp. 363–374 (2015)
13. Mariappan, M., Vora, K.: Graphbolt: dependency-driven synchronous processing of streaming graphs. In: Proceedings of the Fourteenth EuroSys Conference 2019. EuroSys (2019)
14. Nilakant, K., Dalibard, V., Roy, A., Yoneki, E.: Prefedge: SSD prefetcher for large-scale graph traversal. In: Proceedings of International Conference on Systems and Storage, pp. 1–12. SYSTOR (2014)
15. Ren, C., Lo, E., Kao, B., Zhu, X., Cheng, R.: On querying historical evolving graph sequences. VLDB, 726–737 (2011)

16. Shun, J., Blelloch, G.E.: Ligra: a lightweight graph processing framework for shared memory. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 135–146 (2013)
17. Sikos, L.F., Philp, D.: Provenance-aware knowledge representation: a survey of data models and contextualized knowledge graphs. Data Sci. Eng. **5**(3), 293–316 (2020)
18. Then, M., Kersten, T., Günnemann, S., Kemper, A., Neumann, T.: Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs. VLDB, 877–888 (2017)
19. Toss, J., Pahins, C.A.L., Raffin, B., Comba, J.L.D.: Packed-memory quadtree: a cache-oblivious data structure for visual exploration of streaming spatiotemporal big data. Comput. Graph. **76**(NOV.), 117–128 (2018)
20. Wu, H., Zhao, Y., Cheng, J., Yan, D.: Efficient processing of growing temporal graphs. In: DASFAA, pp. 387–403 (2017)
21. Yang, J., Yao, W., Zhang, W.: Keyword search on large graphs: a survey. Data Sci. Eng. **6**(2), 142–162 (2021)
22. Ying, T., Chen, H., Jin, H.: Pensieve: skewness-aware version switching for efficient graph processing. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 699–713 (2020)
23. Zuckerberg, M.: Facebook (2004). http://www.facebook.com