Big Data Mining and Analytics

Volume 2 | Issue 1 Article 5

2019

Auxo: A Temporal Graph Management System

Wentao Han
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China.

Kaiwei Li
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China.

Shimin Chen
Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China.

Wenguang Chen
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China.

Recommended Citation

Wentao Han, Kaiwei Li, Shimin Chen et al. Auxo: A Temporal Graph Management System. Big Data Mining and Anyalytics 2019, 2(1): 58-71.

BIG DATA MINING AND ANALYTICS
ISSN 2096-0654 06/06 pp58–71
Volume 2, Number 1, March 2019
DOI: 10.26599/BDMA.2018.9020030

Auxo: A Temporal Graph Management System

Wentao Han*, Kaiwei Li, Shimin Chen, and Wenguang Chen

Abstract: As real-world graphs are often evolving over time, interest in analyzing the temporal behavior of graphs has grown. Herein, we propose Auxo, a novel temporal graph management system to support temporal graph analysis. It supports both efficient global and local queries with low space overhead. Auxo organizes temporal graph data in spatio-temporal chunks. A chunk spans a particular time interval and covers a set of vertices in a graph. We propose chunk layout and chunk splitting designs to achieve the desired efficiency and the abovementioned goals. First, by carefully choosing the time split policy, Auxo achieves linear complexity in both space usage and query time. Second, graph splitting further improves the worst-case query time, and reduces the performance variance introduced by splitting operations. Third, Auxo optimizes the data layout inside chunks, thereby significantly improving the performance of traverse-based graph queries. Experimental evaluation showed that Auxo achieved $2.9\times$ to $12.1\times$ improvement for global queries, and $1.7\times$ to $2.7\times$ improvement for local queries, as compared with state-of-the-art open-source solutions.

Key words: graphs and networks; temporal databases; composite structures

1 Introduction

Graphs are an important data model for representing complex relationships in big data applications. Representative real-world graph applications include the Web, social networks, road networks, and semantic networks. As real-world graphs are often evolving over time, interest in the temporal behavior of graphs has increased. *Temporal graph analysis* usually accesses a series of snapshots of a graph over time, and then either performs iterative computation over the full snapshots or visits individual vertices. Recent studies analyzed the evolution of the ranks of web pages[1], investigated the impact of user activities on social relationships in social networks[2], and characterized the changes of graph diameters in social networks[3]. Temporal graph analysis is becoming an increasingly significant approach in enhancing static graph analysis and revealing the dynamic time-evolving properties of graphs.

A system to support graph analysis comprises a graph computation engine and a graph management system. Temporal graph analysis requires re-designing both components. For computation engines, recent work has proposed Chronos, an in-memory temporal graph engine[4] that exploits locality-aware batch scheduling to speed up the computation of each vertex across multiple graph snapshots. In this study, we focus on designing a temporal graph management system that efficiently handles the storage and retrieval of evolving graph structures.

Previous work on DeltaGraph proposed a tree-like structure to store an evolving graph[5]. Conceptually, the leaf nodes of the tree represent equi-spaced

• Wentao Han, Kaiwei Li, and Wenguang Chen are with Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: hanwentao@tsinghua.edu.cn; lkw17@mails.tsinghua.edu.cn; cwg@tsinghua.edu.cn.
• Shimin Chen is with Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China. E-mail: chensm@ict.ac.cn.
∗ To whom correspondence should be addressed.
Manuscript received: 2018-05-08; accepted: 2018-05-27

historical snapshots of the graph. A non-leaf node represents a particular intermediate state (e.g., intersection) of all its child nodes. Interestingly, DeltaGraph does not actually store the nodes. Instead, it stores data for each edge in the tree, which contains a set of update events in the evolving graph and represents the delta (i.e., set difference) between the two tree nodes connected by the tree edge. To obtain a graph snapshot, DeltaGraph computes a path from the root to the snapshot, and merges all the tree edge deltas along that path.

However, DeltaGraph has the following main limitations. First, DeltaGraph may incur significant overhead for retrieving graph snapshots. The query time is proportional to the total number of update events regardless of the specific time points of the snapshot to be queried. This means that accessing an early snapshot A of a graph would incur similar cost as compared to accessing a recent snapshot B of the graph, even if A is much smaller than B. Second, to retrieve information of a vertex, DeltaGraph has to combine the data from multiple edges, incurring a significant number of random I/Os. Third, DeltaGraph requires the computation of the difference function at the non-leaf nodes for incoming update events, which may incur non-trivial overhead. Finally, DeltaGraph does not target graph specific features. For example, it does not consider neighborhood relationships, which is important for friend-of-friend queries (also knwon as "2-hop queries").

In this study, we propose Auxo, a novel temporal graph management system that divides and manages temporal graph data in spatio-temporal chunks (Auxo is the goddess of growth in the Greek mythology.). A chunk spans across a time interval and covers a set of vertices in the graph. By judiciously choosing the time points to split the data, for naturally growing graphs, Auxo can achieve both provably linear space complexity for storing temporal graph data and provably linear time complexity for global snapshot queries. By splitting the data in the vertex dimension, Auxo can improve the worst-case query performance and reduce performance variance caused by chunk splitting operations. Auxo employs a neighborhood-aware vertex layout in the chunks to further improve the performance of 2-hop queries. Experimental evaluation has demonstrated that Auxo achieves 2.9× to 12.1× improvements for global queries, and 1.7× to 2.7× improvements for local queries, as compared with state-of-the-art open-source solutions.

The contributions of this article are three-fold. First, we propose Auxo as a novel temporal graph management system that exploits spatio-temporal chunks to efficiently store and retrieve temporal graph data. Second, we study and analyze the use of spatio-temporal chunks as a key technique for temporal graph management. We propose an adaptive exponential split technique and employ a neighborhood-aware vertex layout to achieve efficient queries and space usage. Third, we present a real-machine performance study that compares Auxo with multiple state-of-the-art solutions, using representative graph stores, relational databases, key-value stores, and the most recent temporal graph storage solution.

The remainder of this article is organized as follows. Section 2 forms the discussion by describing the temporal graph model and the programming interface. Then Section 3 presents the design of Auxo, and Section 4 analyzes spatio-temporal chunks. After that, Section 5 empirically evaluates Auxo. Section 6 discusses related work. Finally, Section 7 presents the conlusion of this study.

2 Temporal Graph

We start our discussion by describing the model and the programming interface of temporal graphs in this article.

2.1 Model of temporal graph

A static graph $G = (V, E, P)$ comprises a set V of vertices, a set E of edges, and a set P of properties. Without loss of generality, we consider only directed graphs, where each element $e \in E$ is an ordered pair (u, v). Note that an undirected graph can be represented by replacing every undirected edge with two directed edges. A property $p = (x, key, value)$ specifies a key-value pair for a vertex or an edge as identified by x.

A temporal graph is a series of graph snapshots,

$$\mathbb{G} = \langle G_0, G_1, G_2, \ldots, G_t, \ldots \rangle,$$

where $G_t = (V_t, E_t, P_t)$ is the *snapshot* static graph of $\mathbb{G}$ at time point t, assuming that time is discrete and starts at 0. This is called the *snapshot representation* of a temporal graph.

When a vertex v is added to $\mathbb{G}$ at time t_a, it is assigned an id. This id uniquely identifies v during its lifetime and even after its removal. The id will never be reused. Suppose v is removed at time t_r, then v is *alive*

during the time interval $[t_a, t_r)$. That is, $v \in V_t$, where $t_a \leqslant t < t_r$. This definition follows the transaction time model of temporal records in Ref. [6]. Similarly, an edge is uniquely identified with an id and is alive during an interval (Unique edge ids are capable of representing multiple edges from u to v.). We use a specific bit in an id to distinguish a vertex from an edge. A property is alive from the set operation of the property till the next set or removal operation of the same key for the same id.

A temporal graph can also be defined as a sequence of events:

$$\mathbb{G} = \langle \mathrm{ev}_1, \mathrm{ev}_2, \mathrm{ev}_3, \ldots \rangle,$$

where ev_i $(i = 1, \ldots)$ are events in the following categories:

- (AV, v, t): addition of a vertex, where v is the id of the vertex to add and t is the time of this event;
- (RV, v, t): removal of a vertex v at time t;
- $(\mathrm{AE}, e, u, v, t)$: addition of an edge, where e is the id of the edge to add, u (v) is the source (target) vertex id of e, and t is the time of this event;
- (RE, e, t): removal of an edge e at time t;
- $(\mathrm{SP}, x, key, value, t)$: set of a property of a vertex or an edge as identified by x, where *key* and *value* are the key and the associated value to be set, and t is the time of this event;
- (RP, x, key, t): removal of x's property *key* at time t.

This is called the *event representation* of a temporal graph.

Figure 1 shows an example of a temporal graph. There are 5 snapshots in this example. The first snapshot G_0 is empty. The events are listed below every snapshot.

Every event of a temporal graph is *associated* with an object. The object is a vertex, an edge, or a property of a vertex or an edge. So every event is associated with a vertex. For a vertex event, the vertex itself is the associated vertex of the event. For an edge event, the associated vertex is the source vertex. And for a property event, the associated vertex is the vertex or the source vertex of the edge, depending on the owner of the property.

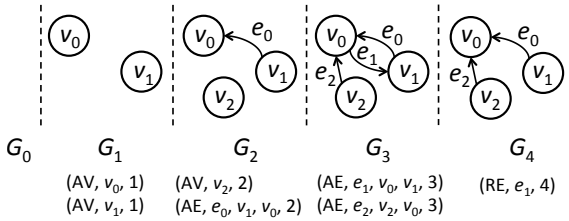


Fig. 1 The evolution of a sample social network.

2.2 Query types

Auxo supports both global queries and local queries:

- *Global queries*: Very common in temporal graph computation[4, 7], a global query visits all the vertices and edges in the snapshot at a given time t.
- *Local queries*: A local query visits a vertex v's edges, and/or its neighbors at a given time t. Only vertices or edges that are alive at time t will be returned by the queries. More complex temporal graph queries can be constructed using local queries.

3 System Design

We present the design of Auxo in this section. After discussing the design goals and challenges, we describe the main technique and the storage layout of our design in detail.

3.1 Design goals and challenges

We consider the following three goals in the design of Auxo:

- *Efficient global queries*: A global query reads all the data for a specified snapshot of the graph. A good design will perform sequential I/Os. Reducing the I/O size requires to store the relevant data of a snapshot as close on disk as possible and to reduce the interleaving of irrelevant data and relevant data.
- *Efficient local queries*: A good design will reduce the number of random I/Os for local queries. For the popular 2-hop queries, it would be nice to place the vertices in a way so that neighbors are more likely to be found on the same disk page.
- *Low space overhead*: The same data (or events) may be stored multiple times in order to improve query performance. However, a good design should bound the space overhead. While the cost of disk space may be less of a concern today, smaller disk footprints will lead to better query performance when main memory cache is used. This is because larger fractions of the data on disk can be cached given the same memory size. Moreover, lower space overhead also means lower amount of data to generate and write to disk, and thus lower overhead for writing data.

Let us consider two naïve extreme cases (as discussed by Salzberg and Tsotras[6]) to understand the challenges. The copy approach stores the snapshot at every time point. It is optimal for global and local

queries. However, since only a very small portion of the graph is usually updated at every time point, the snapshots are highly redundant, incurring intractable space overhead. In comparison, the log approach stores every event just once in the time order. It minimizes space overhead. But global queries have to read the entire log up to the query time point, and local queries will need to perform a random I/O for each event relevant to the given vertex. In general, one can combine copy and log into a hybrid approach. However, how to find a sweet spot in the design space to achieve the three design goals for temporal graphs remains a significant challenge, which we aim to address.

3.2 Spatio-temporal chunks

We propose to store temporal graph data in basic logic units, called *spatio-temporal chunk*s (referred to as chunks for short). A chunk is a region in the structure-time plane, spanning a time interval and covering a set of vertices. Formally, a chunk $C = (V_c, [s_c, t_c))$, where V_c is a set of vertices and $[s_c, t_c)$ is a time interval. For example, as shown in Fig. 2, Chunk C_1 covers all the events that are associated with $v \in V_1$, and are alive in the time interval $[s_1, t_1)$. In this way, a temporal graph can be represented by a number of disjoint chunks, the union of which covers the entire history and all the vertices in the graph.

As shown in Fig. 2, there are two types of chunks. If $t_c < now$, the chunk is *sealed*. A sealed chunk cannot be modified. In comparison, new events will be appended to the chunks on the rightmost side of the plane, where $t_c = now$. These chunks are called *unsealed* chunks. The top-most unsealed chunk is special (e.g., U_4 in Fig. 2). Events on new vertices will be appended only to this special chunk. The vertex set of any other chunk remains unchanged. The process of turning an unsealed chunk into a sealed chunk is called *sealing*. When sealing a chunk, we have the opportunity to optimize the data organization in the chunk for better time and space cost.

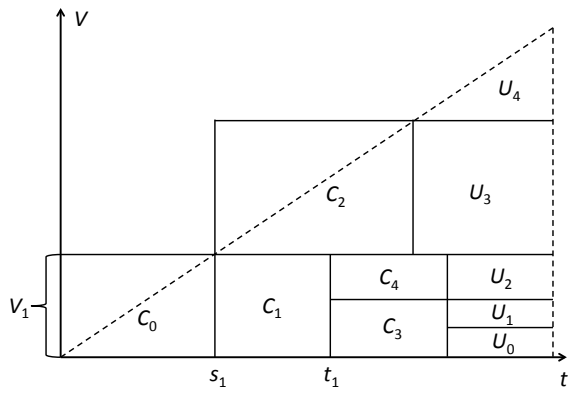


Fig. 2 Chunks tile to cover the entire temporal graph.

3.3 System overview

Figure 3 illustrates a complete temporal graph processing system. At the core is a Temporal Graph Management System (TGMS). TGMS organizes the temporal graph data and supports the temporal graph operations, including vertex/edge/property addition/removal, and global and local temporal queries.

The figure also depicts two other components. Ingester retrieves events from the outside and deals with out-of-order events by buffering and sorting the events. It delivers the events to TGMS in non-decreasing time order. Query Engine communicates with users and parses user queries. It executes temporal queries by invoking methods in the temporal graph interface provided by TGMS.

In this article, we focus on the design of a TGMS—Auxo. Auxo organizes temporal graph data in spatio-temporal chunks, and maintains the metadata of all the chunks. In the following, we describe (a) how to lay out data in chunks and how to answer queries (Section 3.4), (b) how to create new chunks (Section 3.5), and (c) when to create new chunks (Section 4).

3.4 Storage layout

Recall that sealed chunks are immutable and unsealed chunks are appendable. We propose different storage layouts for sealed vs. unsealed chunks. Moreover, we introduce a global index for quickly locating the information of a vertex at a specified time point.

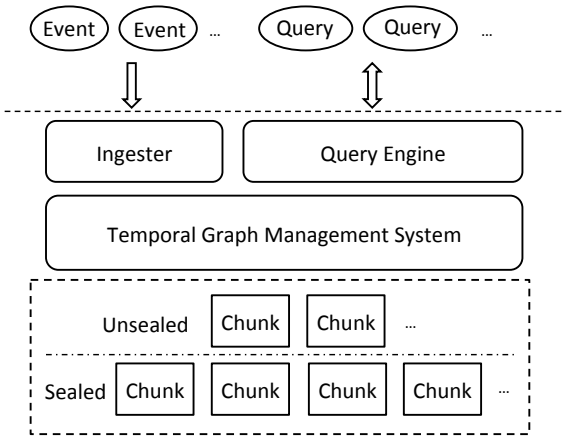


Fig. 3 The overview of a temporal graph processing system.

3.4.1 Sealed chunk layout

A sealed chunk $C = (V_c, [s_c, t_c))$ $(t_c < now)$ covers (1) the events relevant to V_c that are alive at s_c, which are actually the snapshot of V_c at s_c and (2) the relevant events that occur during $[s_c, t_c)$, which are essentially the log during time interval $[s_c, t_c)$.

Figure 4 shows the data layout of a sealed chunk. It consists of vertex segments and a chunk index. A vertex segment consists of the events of a vertex v, followed by the events of v's out-edges. We use 64-bit integers for all fields unless otherwise noted.

• Vertex/edge id: The three most significant bits of the unique id are reserved, indicating whether the event is for a vertex or an edge, whether it is an addition or a removal, whether it is about an entity (i.e., a graph structural change on the vertex or the edge itself) or about a property. Moreover, an edge id contains its source vertex id as prefix.

• Time: the UNIX time in milliseconds when the event occurs.

• Target id (optional): the target vertex id of an added edge.

• Data (optional): 32-bit key size, 32-bit value size, a key, and a value that describe the property. The key and the value are variable sized byte arrays padded to multiples of 8 bytes.

A sealed chunk $C = (V_c, [s_c, t_c))$ will be used in queries involving any vertices in V_c and time point within $[s_c, t_c)$.

For a global query at time $t \in [s_c, t_c)$, the entire chunk is scanned sequentially. It skips any event whose time is greater than t. Moreover, for a vertex, an edge, or a property, the algorithm emits only the last event whose time is less than or equal to t because earlier events are overridden by the last one. Furthermore, if the last event is a removal event, nothing will be emitted because the object is removed as of t.

If a local query for a vertex v at time t belongs to C, we know that all the relevant data are contained in a single vertex segment in the chunk. Therefore, a local query requires only a single random I/O to access the vertex segment (besides the I/Os to search the global index, as will be detailed in Section 3.4.3). Our sealed chunk layout minimizes the I/O cost for local queries.

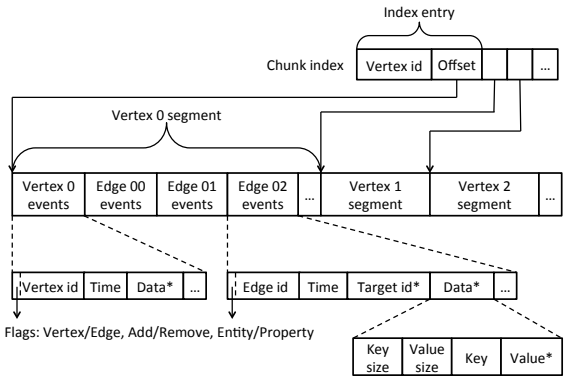


Fig. 4 Layout of a sealed chunk. Fields with asterisks are optional.

3.4.2 Unsealed chunk layout

An unsealed chunk U logically contains events of a vertex set V_u that are either alive at time s_u or happen at or after time s_u. Figure 5 shows the data layout of an unsealed chunk. We employ a design similar to the SSTable in Bigtable[8], which is a variant of the Log-Structured Merge-Tree[9]. First, we store a snapshot at s_u for U using the same data layout as the sealed chunk. Note that the special unsealed chunk that accepts new vertices does not store the snapshot since there is no event at the beginning. Second, new events are inserted into an in-memory event table, as well as appended to a log buffer on disk. Third, when the in-memory table grows to a threshold, Auxo persists it to disk as a log *piece*. The log piece also employs the layout of a sealed chunk. Finally, when there exist k log pieces of the same size, they are merged into a new (level-2) larger piece. Once the level-2 piece is written, the original k pieces are discarded. Similarly, if there exist k level-j pieces, they are merged into a level-$(j+1)$ piece. This merge operation is done by k-way merge.

For a query at time t on U, we need to visit the snapshot, the log pieces that overlap $[s_u, t]$, and the in-memory table if it contains events that occur in $[s_u, t]$. For a global query, we also use a merging algorithm to compute the snapshot. For a local query, we get the offsets of vertex segments in all the relevant components (snapshot, log pieces, and/or in-memory table), and then merge these vertex segments to obtain the result.

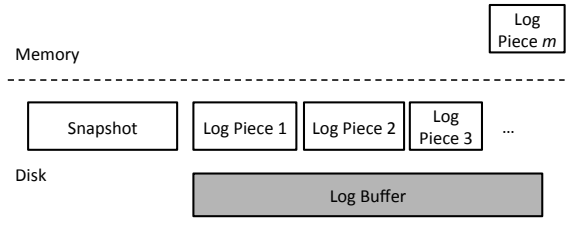


Fig. 5 The physical layout of an unsealed chunk. The log buffer (in gray) has no index.

From this discussion, it is easy to see the benefits of the log piece merging approach. First, it limits the number of log pieces to be merged for a global query. Second, a local query potentially needs to perform an I/O for each log piece to collect the relevant vertex segments. Reducing the number of log pieces reduces the number of random I/Os.

3.4.3 Global index

We design a global index to quickly locate data for local queries. Given a vertex v and time t, the global index finds which chunk and at what position inside the chunk the relevant vertex segment is stored. Figure 6 shows the global index structure, which is similar to the structure of chunks. Each entry in the global index consists of a vertex id, the time when the chunk is generated, a chunk id, and an offset in the chunk. When an unsealed chunk is sealed, entries for the newly generated sealed chunk are inserted, and entries for the unsealed chunk are removed. There can be multiple sealed global indexes, and an unsealed global index. The unsealed global index will be sealed when there are too many entries with the same vertex id. For each vertex, the last index entry in the unsealed global index always refers to an unsealed chunk.

3.5 Chunk splitting

As new events continue to arrive, the size of an unsealed chunk becomes larger and larger. In order to organize data in units of tractable sizes and achieve good space and time efficiency, we will split unsealed chunks. Conceptually, our splitting mechanisms are similar to the time-split and the key-split operations in TSB-tree[10]. However, a TSB-tree organizes data into pages with keys sorted in a page. In contrast, Auxo organizes events in chunks, whose size may vary. There is no clear order of vertices in a graph.

For a temporal graph, we design two types of split operations: time split and graph split. A time split operation splits an unsealed chunk into a sealed chunk and another unsealed chunk that covers the same vertex set, e.g., C_2 and U_3 in Fig. 2. A graph split operation splits a chunk into two. For simplicity, Auxo performs graph split only when doing time split. A time+graph split operation will turn an unsealed chunk into a sealed chunk and two unsealed chunks, e.g., C_3, U_0, and U_1 in Fig. 2.

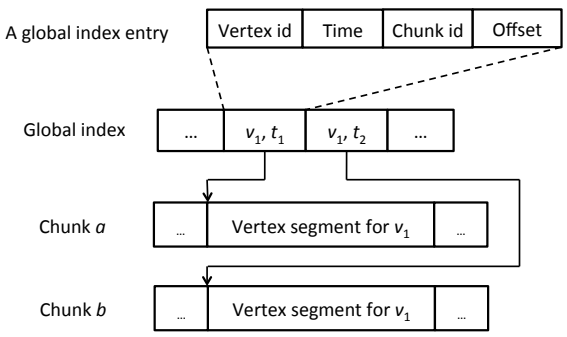


Fig. 6 The global index.

3.5.1 Time split

When an unsealed chunk grows too large, it will become inefficient for temporal queries. First, for a global query, it is possible that a lot of early events have already been updated by some later events, and will not appear in the result of the query. These events are irrelevant but still need to be scanned, which makes the query less efficient. Second, a local query needs to perform a random I/O for the snapshot, and a random I/O for every relevant log piece that overlaps the query's time point as described previously, incurring significantly more random I/Os than a sealed chunk. The number of random I/Os increases as the size of the unsealed chunk increases.

We perform time split to tackle this issue. When doing a time split on an unsealed chunk U, the split is always at the current time point. Let t be the current time at the start of the time split. That is, the operation seals U into a sealed chunk C, and starts a new unsealed chunk U'. For sealing U, we merge U's snapshot, all log pieces, and the in-memory table. For the new unsealed chunk U', all the events that are not alive at t are removed. This often significantly reduces the size of the unsealed chunk. We obtain a snapshot of U at t using a global query, and store the snapshot for U'. From now on, new events will be written to U'.

To avoid the interference between a time split and new events that arrive during the split, new events are appended into a new log piece (which belongs to U'). When the system finishes creating C and U', it must atomically switch U to C and U'. During the transition, the global index is locked. New index entries representing the addition and removal of vertex segments are inserted. Now the global index points to the new sealed chunk C or the new content of the unsealed chunk U', and the old content of U can be discarded. After the time split action, queries at an earlier time than t will be executed on chunk C. Now chunk C does not contain any events that happen at or

after time t. And queries at a more recent time will be executed on the new unsealed chunk U', in which the events not alive after t have already been removed. Hence, by doing time split actions, we improve the performance of query operations.

The time points at which time split actions should be taken are very important to the performance. We will discuss how to decide the split points in Section 4.1.

3.5.2 Graph split

Only doing time split on temporal graphs is not enough for real applications. Take a growing-only temporal graph $\mathbb{G}$ as an example, that is, new vertices and edges are constantly added to the graph, but no one is removed. The snapshot size becomes larger and larger when $\mathbb{G}$ evolves, so do the chunks sealed. It will take a long time to seal a big chunk, and if we can not control the time cost for a single sealing operation, the whole system will suffer a burst at times.

To solve this problem, we also split the graph structure. For example, when the snapshot $G_t = (V_t, E_t)$ reaches a size threshold, we can split V_t into two disjoint sets V_1 and V_2. Then, all the events associated with vertices in V_1 can be organized in one chunk, while all the events associated with vertices in V_2 in another. Once the graph is split, each new chunk is growing by itself. Now the cost of split is controlled under a threshold, and amortized over the entire growing process.

To simplify the design, we only do graph split when doing time split. When doing time split, we further check whether the size of the new unsealed chunk to generate is greater than the chunk size threshold. If so, two unsealed chunks will be generated instead of one, each contains about half of the events.

When doing graph split, we actually partition the graph into two parts by vertices. To optimize graph traversal queries, we want neighboring vertices to be placed in the same chunk as many as possible. Mature graph partitioning algorithms like METIS[11] are used here, since they fulfill our requirement well.

3.5.3 Graph rearrangement

For graph data, traversal is a common pattern of queries. In traversal queries, the data of a vertex is visited, followed by the visit of its edges and neighbors. Take 2-hop neighborhood query as an example. In such a query, we are given a vertex v_*, and want to visit its neighbors as well as the neighbors of its neighbors.

Our design of time split and graph split creates a good opportunity to improve the data locality after the data have already been stored in the system for traverse-based queries. The graph data are eventually stored on disk, organized in blocks. As Fig. 7 shown, for an identical graph structure, different orders of vertices may result in different numbers of blocks to visit. To traverse v_4 in this extracted subgraph, suppose just the data of v_4, v_1, and v_9 need to be visited. Then, for Order 1, the data of these 3 vertices are placed together and in a single disk block, so only one block is needed. For Order 2, the data of these 3 vertices are placed separately in 3 different blocks, so we must visit 3 blocks for the same query. This will cause different performance for queries.

When generating sealed chunks, we can rearrange the order of vertices on the disk to improve the locality for traversal queries. For graph traversal queries like 2-hop neighborhood, we want the data of adjacent vertices to be placed in consecutive storage addresses. It is an NP-hard problem to compute the optimal order of a graph in a line[12]. We can use low-time cost heuristic approach like BFS to achieve a relative good result.

Since we have an index table for vertices in a sealed chunk, what is needed for rearrangement is computing merely a vertex order mapping, describing which vertex should be placed at which position, and then generating the sealed chunk in this order.

4 Choosing Policy

The design of spatio-temporal chunks minimizes the number of random disk accesses for local queries. In this section, we analyze the time cost for global queries and the space cost of our design in order to guide Auxo to choose time split and graph split policies.

4.1 Time split policy

For simplicity, we assume an ideal case where the graph is growing constantly and uniformly, and the size of events is the same (and is normalized to 1). Let a be the rate of addition events, r be the rate of removal events, and u be the rate of update events. If we use

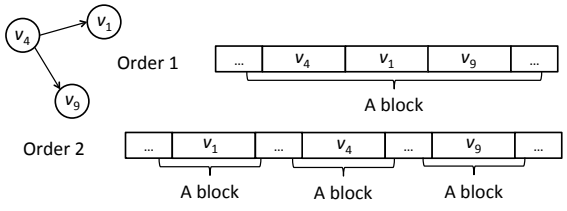


Fig. 7 Two different layouts of a subgraph.

the *event order number* as time, then $a + u + r = 1$. Let $\alpha = a - r$ be the net growing rate. Since the graph is growing, $\alpha \geqslant 0$. Let β be the size of the snapshot at the very beginning. Thus, at time τ, the total number of new events is τ, and the total number of live events is $\beta + \alpha\tau$.

Let τ_n be the time when the n-th time split is taken ($\tau_0 = 0$). The chunk consists of the live events at time τ_{n-1} and all the events between τ_{n-1} and τ_n. Therefore, the chunk contains $(\beta + \alpha\tau_{n-1}) + (\tau_n - \tau_{n-1})$ events.

We define two factors to measure the space and time overhead of the design. The space factor SF_n at the n-th time split is defined as the space taken by all the chunks divided by the space of all events (excluding the snapshot of size β at the beginning):

$$\mathrm{SF}_n = \frac{\sum_{k=1}^{n} [(\beta + \alpha\tau_{k-1}) + (\tau_k - \tau_{k-1})]}{\tau_n} = 1 + \frac{n\beta + \alpha \sum_{k=1}^{n} \tau_{k-1}}{\tau_n}.$$

It is necessary to store all the events to support temporal queries. Therefore, $\mathrm{SF}_n = 1$ is the best possible case with no space overhead. Space overhead arises because an event may be stored in multiple chunks. Since the total chunk size corresponds to the I/O cost for generating time splits, SF_n also reflects the overhead of the time split operations.

The time factor TF_n of the n-th chunk is defined as the size of the n-th chunk divided by the size of the total live events at τ_{n-1}:

$$\mathrm{TF}_n = \frac{(\beta + \alpha\tau_{n-1}) + (\tau_n - \tau_{n-1})}{\beta + \alpha\tau_{n-1}} = 1 + \frac{\tau_n - \tau_{n-1}}{\beta + \alpha\tau_{n-1}}.$$

The global query at time $\tau \in (\tau_{n-1}, \tau_n]$ is answered by reading the n-th chunk, whose size is the numerator in TF_n. However, the best possible time to answer such a query is to read only the live events, whose size is $\beta + \alpha\tau$. This is lower bounded by $\beta + \alpha\tau_{n-1}$, which is the denominator of TF_n. Therefore, TF_n represents the worst-case time overhead for a global query.

Table 1 shows the space and time factors by setting the split time τ_k to be a polynomial or an exponential function of k. From the analysis, we see that there is a trade-off between space and time overheads. Smaller TF_n often means larger SF_n.

When $\alpha > 0$, for the polynomial functions, TF_n goes to 1 for sufficiently long time, but SF_n is not bounded by a constant. On the other hand, the exponential functions lead to constant SF_n and TF_n asymptotically. In particular, when $b = 1 + \alpha$, SF_n and TF_n both become 2. Therefore, exponential time splits achieve more balanced space and time overheads.

When $\alpha = 0$, exponential time split is a poor choice. It will lead to larger and larger chunks, while the total number of live events remains constant. TF_n will grow to infinity. In contrast, the linear split policy with $\Delta = \beta$ achieves constant space and time factors; both SF_n and TF_n are 2.

Combining the analysis in the above two cases, we propose the *Adaptive Exponential Split* policy. Suppose S_U is the size of the last snapshot, and L_U is the size of the current log in an unsealed chunk U. We perform a time split when $\frac{L_U}{S_U} \geqslant \lambda$ and $L_U \geqslant \gamma$, where λ is the base parameter, and γ is the split threshold. For the ideal constantly growing case, $\lambda = \frac{b-1}{\alpha}$. λ tunes the space-time trade-off. This policy is able to achieve constant space and time factors for both $\alpha > 0$ and $\alpha = 0$ cases.

We compare our solution to DeltaGraph[5]. Note that while the DeltaGraph paper considers the Copy+Log approach, it has not studied the exponential split policy. First, DeltaGraph with the Balanced function requires $O(N \log N)$ space, where N is the total number of events till now (Here N is equivalent to $|E|$ in DeltaGraph's notation.). In comparison, our solution achieves $O(N)$ space complexity, which significantly reduces the space overhead. Second, for retrieving a graph snapshot, DeltaGraph's time cost is $O(N)$

Table 1 Space and time factors of different time split policies. N=τ_n stands for total number of events. Δ is a constant controlling the time split behavior.

τ_k	SF_n	$\mathrm{SF}_n, N \to +\infty$	Total space	TF_n	$\mathrm{TF}_n, N \to +\infty$
$k\Delta$	$1 + \frac{\beta}{\Delta} + \alpha\frac{n-1}{2}$	$\frac{\alpha}{2}\frac{N}{\Delta}$	$O(N^2)$	$1 + \frac{1}{\beta/\Delta+\alpha(n-1)}$	1
$k^2\Delta$	$1 + \frac{\beta}{n\Delta} + \alpha\left(\frac{n}{3} - \frac{1}{2} + \frac{1}{6n}\right)$	$\frac{\alpha}{3}\sqrt{\frac{N}{\Delta}}$	$O(N^{\frac{3}{2}})$	$1 + \frac{2n-1}{\beta/\Delta+\alpha(n-1)^2}$	1
$k^p\Delta$, $p \in \mathbb{Z}$, $p \geq 1$	$1 + \frac{\beta}{n^{p-1}\Delta} + \alpha\left(\frac{n}{p+1} + \cdots\right)$	$\frac{\alpha}{p+1}\sqrt[p]{\frac{N}{\Delta}}$	$O(N^{\frac{p+1}{p}})$	$1 + \frac{pn^{p-1}+\cdots}{\beta/\Delta+\alpha(n-1)^p}$	1
$2^k\Delta$	$1 + \frac{n\beta}{2^n\Delta} + \alpha\frac{2^n-1}{2^n}$	$1 + \alpha$	$O(N)$	$1 + \frac{1}{\beta/(2^{n-1}\Delta)+\alpha}$	$1 + \frac{1}{\alpha}$
$b^k\Delta$, $b > 1$	$1 + \frac{n\beta}{b^n\Delta} + \frac{\alpha}{b-1}\frac{b^n-1}{b^n}$	$1 + \frac{\alpha}{b-1}$	$O(N)$	$1 + \frac{b-1}{\beta/(b^{n-1}\Delta)+\alpha}$	$1 + \frac{b-1}{\alpha}$

regardless of the specific time points of the snapshot to be queried. In contrast, our solution achieves the time complexity of $O(m)$ for a global query, where m is the number of live events at the query time. Note that for a normal growing graph, if the query time point is small, it is often the case that $m \ll N$. Therefore, our solution can significantly reduce the time overhead.

4.2 Graph split policy

If the graph is split into multiple partitions, all partitions will grow independently. If we choose different parameters to arrange the partitions to have the peak time factors at different times, the overall worst-case time factor can be further reduced.

We consider the case with $b = 2$ and $\alpha = 1$ for the ideal constantly growing graph. Without graph splits, both SF_n and TF_n are 2. Now suppose the graph is split into 2 partitions. Partition 1 takes k-th time split at $\tau = 2^k$, and Partition 2 at $\tau = 2^k C$, where $C \in (1, 2)$. It is easy to see that the worst-case time factor of the whole graph is reached either at 2^k or at $2^k C$.

$$\mathrm{TF}' = \frac{2^{k+1} + 2^k C}{2 \times 2^k} = 1 + \frac{C}{2}, \quad \text{at } 2^k;$$

$$\mathrm{TF}'' = \frac{2^{k+1} + 2^{k+1} C}{2 \times 2^k C} = 1 + \frac{1}{C}, \quad \text{at } 2^k C.$$

The overall worst-case time factor will be $\max\{\mathrm{TF}', \mathrm{TF}''\}$. This is minimized when $\mathrm{TF}' = \mathrm{TF}''$, i.e., $C = \sqrt{2}$. In this case, the overall time factor is 1.707, which is better than $\mathrm{TF}_n = 2$ for the graph growing as a whole.

Similarly, if the graph is split into w partitions, the optimal worst-case time factor is achieved when Partition i ($1 \leqslant i \leqslant w$) performs time splits at $2^k 2^{i/w}$. In this case, the worst-case time factor is $\dfrac{1}{(1 - 2^{-1/w})w}$. This value decreases when w increases, and has a limit of $\dfrac{1}{\ln 2} = 1.443$. In fact, when $w = 10$, this value has already dropped below 1.5.

In general, for an ideal constantly growing graph with the base parameter $b > 1$ and the growing rate $\alpha > 0$, we can split the graph into multiple graph partitions to reduce the worst-case time factor. The optimal worst-case time factor approaches $\dfrac{(b-1)(\alpha+b-1)}{\alpha b \ln b}$. The proof is absent due to space constraints.

5 Evaluation

5.1 Experimental setup

Machine Configuration. We run all our experiments on a machine equipped with a 3.4 GHz Intel Core i7-4770 CPU, 32 GB main memory, and a Seagate 7200RPM disk drive, running Ubuntu 14.10. Table 2 lists the detailed configuration.

Solutions to compare. We compare five solutions:

• *Auxo* is our proposed solution. We implemented Auxo in C++. Chunks are stored in files on the local disk.

• *DeltaGraph*[5] uses a tree-like structure to store an evolving graph. We implemented a simplified version of DeltaGraph by modifying and reusing Auxo code. It supports only the Balanced differential function, which is recommended as the preferred function in Ref. [5].

• *Neo4j*[13] is a state-of-the-art graph database. In Neo4j, data are organized by nodes, as well as relationships between nodes. Both nodes and relationships can have properties. We store the addition and removal timestamps of vertices and edges in a temporal graph using properties in Neo4j. We implemented queries with Neo4j's Java API.

• *PostgreSQL*[14] is an open-source relational database system with support for temporal features. We store vertices and edges in a vertex table and an edge table, respectively. We build temporal indices using the temporal extension. The queries are implemented in SQL statements.

• *HBase*[15] is a popular key-value data store with support for multi-versioning and range scans. We store vertices and edges as rows in an HBase table, and record addition and removal events using different versions (i.e., timestamps). We implemented temporal queries using HBase's Java API by specifying time ranges.

Data Sets. We use two temporal graph data sets from KONECT[16], as listed in Table 3. *wiki-fr* is a temporal graph for French articles in Wikipedia. Each

Table 2 Hardware and software configuration used.

Item	Configuration/Version
Processor	Intel Core i7-4770 3.40 GHz (4 cores)
Memory	32 GB
Disk	Seagate 3 TB 7200RPM (64 MB cache)
OS	Ubuntu 14.10
Kernel	Linux 3.16.0-28-generic
C++	LLVM/Clang 3.5.0
Java	OpenJDK 1.7.0_75
Neo4j	2.1.6
PostgreSQL	9.4
HBase	1.0.0

Table 3 Statistics of the temporal graphs used. ($\times 10^6$)

Graph	Number of vertices	Number of edges	Number of events
wiki-fr	2.21	24.4	61.2
dblp	1.31	38.0	39.3

vertex is an article, and edges represent references. *dblp* is a co-authorship graph. Each vertex is an author. If two authors co-authored a paper, then there is an edge between the two corresponding vertices. Since it is an undirected graph, we use two directed edges to represent each undirected edge. Our data sets are comparable to the largest synthetic data set and are about 10 times larger than the real-world data set in previous work[5].

Measurement. For each experiment, we perform 5 runs, and report the average execution time. Before each run, the page cache of the operating system is cleaned. We model two situations: (i) no cache and (ii) cache. For the latter, main memory is available to cache graph data. For the former, we ran another program that allocates and locks most of the free memory, leaving only a few megabytes to the temporal graph solutions.

5.2 Evaluation of Auxo

Time-Split Policy. Time-split policies affect the storage usage and the performance of queries. We generate chunks using different time-split policies and parameters, and measure the execution time of global queries. Graph split is not performed in this experiment.

Figure 8 shows the performance of global queries on wiki-fr graph with different time split policies. We conducted 11 global queries at evenly distributed time points between 2005 and 2010. The figure also shows the ideal curve, for which we generate snapshots that contain only the live events at the query time points, and measure the query performance. Figure 8a shows the results for exponential split with $b = 2$, and adaptive exponential split with $\lambda = 1$. Figures 8b and 8c show the execution times under different parameters. Each bump in the curves corresponds to a time split. The curves have many flat portions because several global queries are answered using the same chunk, thus having similar cost. Note that the time factor is the ratio of the execution time to that of the ideal case.

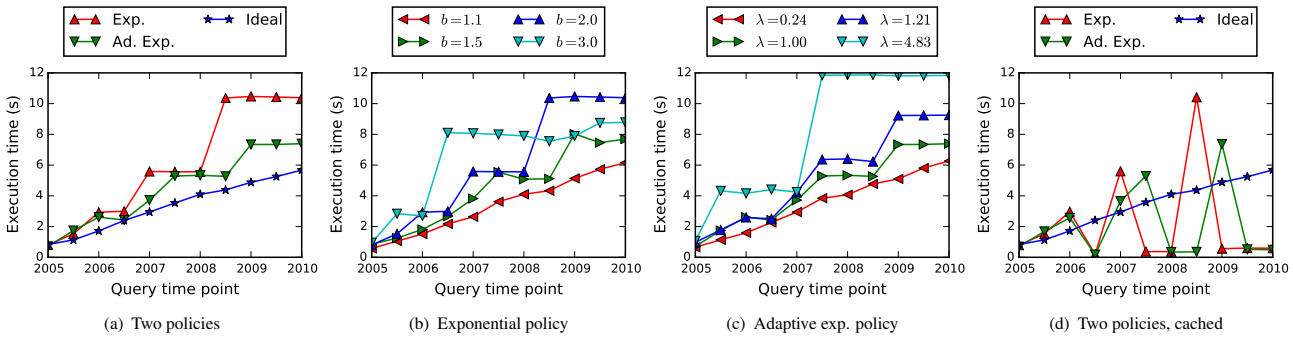
In general, smaller b or λ results in faster query execution (while using larger storage overhead). Interestingly, in Fig. 8b, the curve of $b = 3.0$ is lower than the curve of $b = 2.0$ for global queries at 2008 and later. The reason is that the last time split of the $b = 3.0$ curve is very close to the end of the event series in wiki-fr. If the graph grows infinitely, this anomaly would not happen.

Figure 8d shows the same policies as in Fig. 8a, while having enough memory to cache a single chunk. Hence, if we query several time points in the same chunk, the execution times of the second and later queries to the same chunk are drastically reduced.

Figure 9 shows the same experiment for the dblp graph. Since the co-author relationship is growing only, the two time-split policies have almost the same effects.

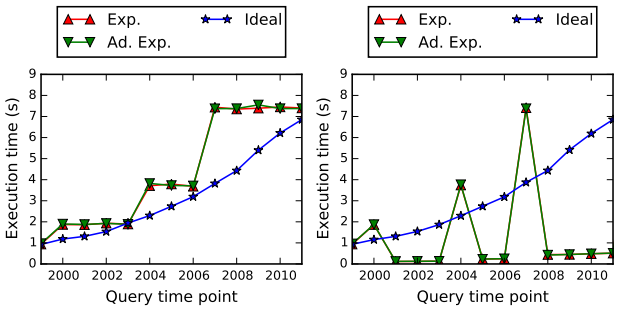
Graph Split. Figure 10 shows the disk write speed over time with and without graph split mechanism when importing dblp graph. For graph split, once the number of new vertices reaches 1×10^5, the special unsealed chunk is sealed. Compared to the run without graph split, though the total sizes of chunks on disk are very close, I/O operations are amortized over time. Hence, graph split can reduce the I/O burst incurred by chunk sealing operations, making the system more predictable.

Graph Rearrangement. Figure 11 shows the effect of graph rearrangement. Before the experiment, we generated a list of 1000 random vertices for each graph, and ran 2-hop neighborhood queries for the same list of vertices with different approaches of rearrangement.



(a) Two policies (b) Exponential policy (c) Adaptive exp. policy (d) Two policies, cached

Fig. 8 Global query performance on wiki-fr graph with different time-split policies and parameters.



(a) No cache (b) Cache

Fig. 9 Global query performance on dblp graph with different time-split policies.

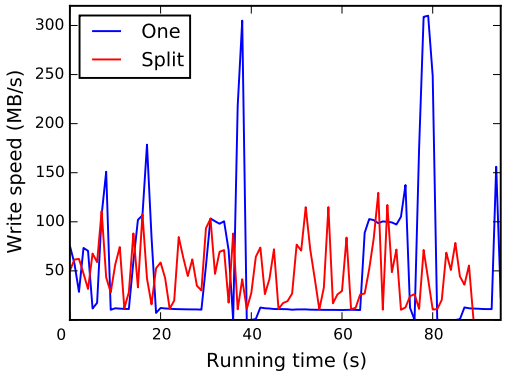
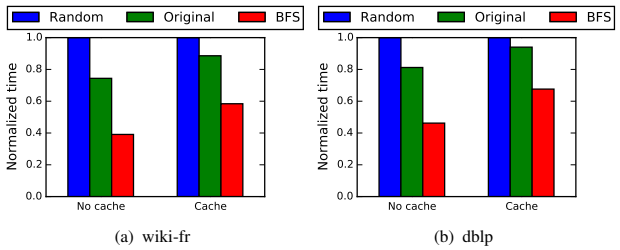


Fig. 10 Disk write speed over time with and without graph split.

For *Random*, the vertices are shuffled randomly (used as a baseline). For *Original*, the vertices are ordered by the sequence that they were added. For *BFS*, the vertices in the chunk are traversed by a breadth-first search, and rearranged by the BFS traversal order. This heuristic has low time cost, and can put adjacent vertices in proximity (Spectral partitioning[17] is another potential approach to rearranging vertices. We have experimented spectral partitioning on small graphs, and found that its computation is very expensive and incurs dramatically high overhead for chunk splits).

Figure 11a shows the results on wiki-fr graph. For each group of bars, the average execution time of Random is normalized to 1. When caching is turned off, *BFS* achieves 61% improvement over *Random*, and 47% improvement over *Original*. When caching is turned on, the performances of all three approaches improve. Before each experiment, we clean the page cache of the OS. When a vertex is accessed, the entire disk page containing the vertex data will be cached in memory. As an experiment proceeds, it is more and more likely that a request for a vertex will hit in the page cache. This effect also smoothes the difference across the three approaches. Nevertheless, *BFS* still sees 42% improvement over *Random*, and 34% improvement over Original.



(a) wiki-fr (b) dblp

Fig. 11 Impact of vertex layout on 2-hop neighborhood queries.

Figure 11b shows the same experiment on dblp. We see similar trends. *BFS* is the best among the three approaches. Since the average degree of dblp is smaller than wiki-fr, the improvement of *BFS* is smaller.

Multithreading. Multiple threads can concurrently perform queries in Auxo. Figure 12 shows the speedup of parallel 2-hop neighborhood queries on wiki-fr. The experimental machine has 4 CPU cores. When we increase the number of threads from 1 to 4, we see a 1.6× improvement in performance. The speedup mainly comes from concurrent access of data cached in memory. However, the concurrency is limited by the IOPS of the single disk in the system.

5.3 Performance comparison with representative state-of-the-art solutions

Comparison to DeltaGraph. Figure 13 shows the performance of global queries on dblp using the Balanced function of DeltaGraph ($L = 10\,000$, $k = 2$) and the adaptive exponential split policy ($\lambda = 1$) of Auxo. The data generated by Auxo is 5.4 GB, while DeltaGraph takes 23 GB. We see that Auxo and DeltaGraph achieve similar query performance for querying relatively late snapshots. On the other hand, for querying early snapshots, DeltaGraph can be up

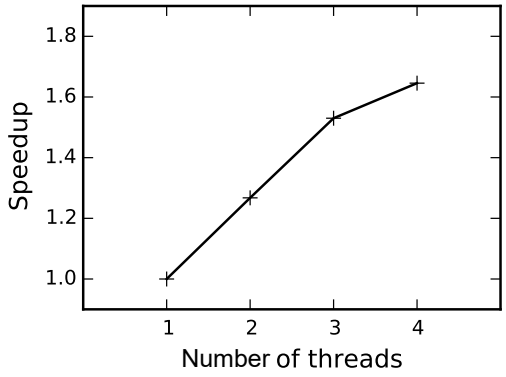


Fig. 12 Multithreaded 2-hop neighborhood queries.

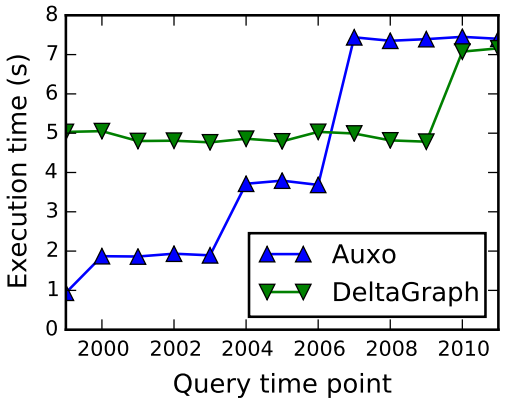


Fig. 13 Comparison of Auxo and DeltaGraph (dblp graph).

to 5× as slow as Auxo because DeltaGraph's query execution time is proportional to the total number of events in the system rather than the actual number of live events at the time of the snapshot.

We also ran 2-hop queries on dblp under the same settings as for global queries. For 1000 randomly generated queries, Auxo took 130 ms on average, and DeltaGraph took 425 ms on average. In local queries, DeltaGraph needs to access blocks containing data of the same vertex across different "edges", while Auxo places the data sequentially in one chunk.

Comparison to Neo4j, PostgreSQL, and HBase. We use default configurations for the three solutions. HBase is running in single-node mode. For global queries, 100 queries with random time points are executed. For 2-hop queries, 1000 queries with random parameters are executed. Table 4 compares the average execution times of Auxo, Neo4j, PostgreSQL, and HBase.

Overall, compared to the three state-of-the-art solutions, Auxo achieves 2.9× to 12.1× improvement for global queries and 1.7× to 2.7× improvement for local queries. For global queries, PostgreSQL is better than Neo4j and HBase due to its temporal indexing support. Auxo is much better than the other three solutions, showing the effectiveness of the spatio-temporal chunks. For 2-hop queries, we are surprised to see that HBase is better than PostgreSQL and Neo4j. Auxo is the best because spatio-temporal chunks minimize the number of random disk accesses for local queries and because of the graph rearrangement optimization.

Table 4 Query performance of different systems.

System	Global (s)	2-hop (ms)
Neo4j	16.0	340
PostgreSQL	6.73	346
HBase	27.9	214
Auxo	2.30	128

6 Related Work

Research on temporal graphs is ongoing, and has found many interesting properties that could not be contained in a single graph snapshot[18–20]. Much research has also proposed algorithms that rely on the dynamics of graphs, and give more profound results[21–23]. Auxo can potentially support the analysis and algorithms of temporal graphs.

DeltaGraph[5] proposes a temporal indexing data structure for temporal graphs, aiming at snapshot retrieval. As we discussed, our design is different from DeltaGraph in the organization method of events, and provides opportunity to optimize graph data locality to improve traverse-based graph query. The system in Ref. [24] also tries to optimize temporal and graph data locality, but it adopts interaction graph model. In this model, each event happens at a time point, but does not span for an interval. So techniques for computational geometry could be employed for this model.

Temporal data access has been studied in relational data model[6]. Data structures like TSB-tree[10] and HV-tree[25] have been proposed to optimize data access for temporal relational databases. ImmortalDB[26,27] puts TSB-tree into reality, and adds new techniques like version compression for saving storage cost and improving query performance. Our design of performing time split and graph split on chunks is inspired by time split and key split in TSB-tree. However, chunks in Auxo are not aligned with physical blocks. The adaptive exponential time split policy keeps both space and time factors as constants. This policy cannot be applied in TSB-tree due to its fixed page size. The flexible size of chunks also provides opportunity for graph layout optimization. Instead of index pages in the tree, we use a global index for fast vertex lookup.

The recent research on graph engines has provided vertex- or edge-centric programming models for iterative graph computation[28–30]. Grace, GraphChi, and X-Stream accelerate graph computation on a single machine[31–33]. They all change the graph layout or the vertex/edge visiting order to reduce the randomness of access from the irregularity of graph structures. Kineograph ingests graph data

from outside, and keeps the most recent snapshot in memory for streaming computation[7]. Chronos assumes temporal graph data stored persistently, and computes general graph algorithms on several snapshots simultaneously[4]. Auxo is complementary to Chronos and could be used as its data source.

7 Conclusion

In this article, we propose a temporal graph storage and query system Auxo. It stores temporal graph data in chunks, and employs an adaptive exponential time-split policy, which constrains both the space and time factors within a constant. Moreover, graph split can help further amortize the time factor, and amortize chunk sealing overhead. Additionally, graph rearrangement in a chunk can improve graph data locality, which optimizes traverse-based queries like 2-hop neighborhood. The overall design makes Auxo an efficient system for temporal graphs.

Acknowledgment

This work was supported by the National High-Tech Development Plan of China (No. 2015AA015306) and the National Natural Science Foundation of China (No. 61772302).

References

[1] L. Yang, L. Qi, Y. Zhao, B. Gao, and T. Liu, Link analysis using time series of web graphs, in *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007*, Lisbon, Portugal, 2007, pp. 1011–1014.

[2] C. Wilson, B. Boe, A. Sala, K. P. N. Puttaswamy, and B. Y. Zhao, User interactions in social networks and their implications, in *Proceedings of the 2009 EuroSys Conference*, Nuremberg, Germany, 2009, pp. 205–218.

[3] J. Leskovec, J. M. Kleinberg, and C. Faloutsos, Graphs over time: Densification laws, shrinking diameters and possible explanations, in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Chicago, IL, USA, 2005, pp. 177–187.

[4] 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 *Proceedings of the Ninth European Conference on Computer Systems EuroSys'14*, New York, NY, USA, 2014, pp. 1:1–1:14.

[5] U. Khurana and A. Deshpande, Efficient snapshot retrieval over historical graph data, in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, 2013, pp. 997–1008.

[6] B. Salzberg and V. J. Tsotras, Comparison of access methods for time-evolving data, *ACM Comput. Surv*., vol. 31, pp. 158–221, 1999.

[7] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, Kineograph: Taking the pulse of a fast-changing and connected world, in *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys'12*, New York, NY, USA, 2012, pp. 85–98.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, Bigtable: A distributed storage system for structured data, *ACM Transactions on Computer Systems*, vol. 26, no. 2, p. 4, 2008.

[9] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, The log-structured merge-tree (lsm-tree), *Acta Inf.*, vol. 33, no. 4, pp. 351–385, 1996.

[10] D. Lomet and B. Salzberg, Access methods for multiversion data, in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data SIGMOD'89*, New York, NY, USA, 1989, pp. 315–324

[11] G. Karypis and V. Kumar, METIS—Unstructured graph partitioning and sparse matrix ordering system, version 2.0, Tech. Rep., 1995.

[12] M. R. Garey, D. S. Johnson, and L. Stockmeyer, Some simplified np-complete problems, in *Proceedings of the Sixth annual ACM Symposium on Theory of Computing*, 1974, pp. 47–63.

[13] Neo4j, http://neo4j.com/, 2015.

[14] PostgreSQL, http://www.postgresql.org/, 2015.

[15] HBase, http://hbase.apache.org/, 2015.

[16] J. Kunegis, Konect: The koblenz network collection, in *Proceedings of the 22nd International Conference on World Wide Web Companion*, 2013, pp. 1343–1350

[17] D. A. Spielmat and S.-H. Teng, Spectral partitioning works: Planar graphs and finite element meshes, in *Proceedings of 37th Annual Symposium on Foundations of Computer Science*, 1996, pp. 96–105.

[18] P. Boldi, M. Santini, and S. Vigna, A large time-aware web graph, *SIGIR Forum*, vol. 42, no. 2, pp. 33–38, 2008.

[19] J. Leskovec, J. Kleinberg, and C. Faloutsos, Graphs over time: Densification laws, shrinking diameters and possible explanations, in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, KDD'05*, New York, NY, USA, 2005, pp. 177–187.

[20] C. Wilson, B. Boe, R. Sala, K. P. N. Puttaswamy, and B. Y. Zhao, User interactions in social networks and their implications, in *Proceedings of the 4th ACM European Conference on Computer Systems*, 2009.

[21] K. Lerman, R. Ghosh, and J. H. Kang, Centrality metric for dynamic networks, in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, 2010, pp. 70–77.

[22] J. Tang, I. Leontiadis, S. Scellato, V. Nicosia, C. Mascolo, M. Musolesi, and V. Latora, Applications of temporal graph metrics to real-world networks, in *Temporal Networks*, 2013, pp. 135–159.

[23] S. Huang, J. Cheng, and H. Wu, Temporal graph traversals: Definitions, algorithms, and applications, arXiv preprint arXiv:1401.1919, 2014.

[24] B. Gedik and R. Bordawekar, Disk-based management of interaction graphs, *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 11, pp. 2689–2702, 2014.

[25] R. Zhang and M. Stradling, The hv-tree: A memory hierarchy aware version index, *Proc. VLDB Endow.*, vol. 3, pp. 397–408, 2010.

[26] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu, Immortal db: Transaction time support for sql server, in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data SIGMOD'05*, New York, NY, USA, 2005, pp. 939–941

[27] D. Lomet, M. Hong, R. Nehme, and R. Zhang, Transaction time indexing with version compression, *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 870–881, 2008.

[28] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, Pregel: A system for large-scale graph processing, in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD'10*, New York, NY, USA, 2010, pp. 135–146.

[29] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, Graphlab: A new framework for parallel machine learning, arXiv Preprint arXiv:1006.4990, 2010.

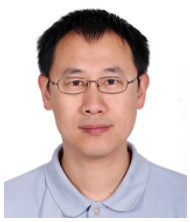
[30] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, Powergraph: Distributed graph-parallel computation on natural graphs, in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12*, Berkeley, CA, USA, 2012, pp. 17–30.

[31] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan, Managing large graphs on multicores with graph awareness, in *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, Boston, MA, USA, 2012, pp. 41–52

[32] A. Kyrola, G. Blelloch, and C. Guestrin, Graphchi: Large-scale graph computation on just a PC, in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12*, Berkeley, CA, USA, 2012, pp. 31–46.

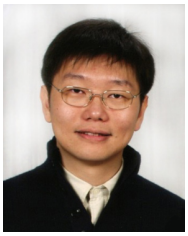
[33] A. Roy, I. Mihailovic, and W. Zwaenepoel, X-stream: Edge-centric graph processing using streaming partitions, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP'13*, New York, NY, USA, 2013, pp. 472–488.

Wentao Han received the bachelor and PhD degrees in computer science from Tsinghua University in 2008 and 2015, respectively. He is currently a postdoc researcher in Department of Computer Science and Technology, Tsinghua University. His research interests include big data processing systems and neuromorphic systems.



Wenguang Chen received the bachelor and PhD degrees in computer science from Tsinghua University in 1995 and 2000, respectively. He was the CTO of Opportunity International Inc. from 2000–2002. Since January 2003, he joined Tsinghua Univeristy. He is now a professor in Department of Computer Science and Technology, Tsinghua University. His research interest is in parallel and distributed computing, programming model, and mobile cloud computing.

Kaiwei Li received the bachelor degree from Tsignhua University in 2014. He is currently a PhD student in Department of Computer Science and Technology, Tsinghua University. His research interest is in parallel computing.



Shimin Chen received the PhD in computer science from Carnegie Mellon University in 2005, and BE and ME degrees from Tsinghua University in 1997 and 1999, respectively. He is currently a professor in Institute of Computing Technology, Chinese Academy of Sciences. His research interests include database systems, big data systems, and computer architecture.