Noname manuscript No.
(will be inserted by the editor)

HiNode: An Asymptotically Space-Optimal Storage Model for Historical Queries on Graphs

Andreas Kosmatopoulos · Kostas Tsichlas · Anastasios Gounaris · Spyros Sioutas · Evaggelia Pitoura

the date of receipt and acceptance should be inserted later

Abstract Most modern networks are perpetually evolving and can be modeled by graph data structures. By collecting and indexing the state of a graph at various time instances we are able to perform queries on its entire history and thus gain insight into its fundamental features and attributes. This calls for advanced solutions for graph history storing and indexing that are capable of supporting application queries efficiently while coping with the aggravated space requirements. To this end, we advocate a purely vertex-centric storage model that is asymptotically space-optimal and more space efficient than any other proposal to date. In addition to space efficiency, the model's purely vertex-centric approach shows great promise with respect to the efficiency and functionality of update and query operations. Furthermore, we make a qualitative comparison with other general methods for graph history storage identifying the pros and cons of our approach. Finally, we implement and incorporate our technique in the G^* parallel graph processing system, we conduct thorough experimental evaluation and we show that we can yield time and space improvements up to an order of magnitude when compared to G^*.

Keywords Historical Queries · Evolving Graphs · Indexing · Space Efficiency

A. Kosmatopoulos (✉) · K. Tsichlas · A. Gounaris
Department of Informatics, Aristotle University of Thessaloniki, Greece
E-mail: akosmato@csd.auth.gr

K. Tsichlas
E-mail: tsichlas@csd.auth.gr

A. Gounaris
E-mail: gounaria@csd.auth.gr

S. Sioutas
Department of Informatics, Ionian University, Corfu, Greece
E-mail: sioutas@ionio.gr

E. Pitoura
Computer Science Department, University of Ioannina, Greece
E-mail: pitoura@cs.uoi.gr

1 Introduction

The past few years have seen a rapid increase of networks that produce a considerable amount of data. Networks, such as citation networks, traffic networks and social networks are, naturally represented as graphs. These graphs are usually dynamic, in the sense that the network they are representing is constantly evolving with vertices and edges being inserted, updated, or deleted. For example, in a graph representing communities in a social network, new friendships may be created or deleted as time evolves, and in a graph representing collaboration between users (such as in co-authorship networks), both users and collaborations between them change over time.

An important challenge that arises with the dynamicity in such networks (and their respective graphs) is the appropriate handling of their history so that we can extract features and properties that characterize the whole (or part of the) timeline of the graph, as opposed to solely its latest state. By doing so, we are able to answer queries such as *"how has the diameter in a group of friends evolved between 2012 and today"* in social networks or *"how has the closeness centrality of author X progressed over time"* in citation networks. In the first case, a decreasing diameter could signify that the members of the community are becoming more connected between them as time progresses while in the second case an increasing closeness centrality could translate to the author becoming more "influential" as more works are published in the field.

A key aspect in effectively tackling the overall problem is the efficient storage of the evolving graph. We call the states of the evolving graph at different time instances *snapshots* and view the graph as an evolving sequence of such snapshots. Queries may involve one or more of these snapshots. We call the set of operations (i.e., vertex/edge insertion, deletions and updates) between two consecutive snapshots *deltas*. A simple approach to solving the problem would be to explicitly store all snapshots separately. This strategy would be oblivious to the evolution of the graph in time which implies a prohibiting space overhead that also affects the time cost of the supported operations. Consider a sparse graph of 10^6 vertices and 10^6 snapshots, with each snapshot corresponding to a small number (tenths) of changes in edges and vertices. By explicitly storing all snapshots one would require space of 10^{12} magnitude while by exploiting the similarities between successive snapshots one would expect to use space of 10^7 magnitude.

Thus, a system's performance could be significantly improved by maintaining an indexing and storage mechanism that exploits the fact that there are vertices and edges that remain unaltered between snapshots in the sequence. Also, due to the large graph sizes, this mechanism should consider that most of the graph data reside within the external memory. Consequently, the efficient design of the indexing and storage components constitutes a crucial step towards mitigating the impact of frequently accessing the external memory during the processing of evolving graph sequences.

Indexing and storage should also be designed by taking into consideration the types of possible queries on evolving graphs. Queries can be distinguished between (i) *local* queries, which require information of a single graph vertex or a few vertices (e.g. degree centrality), and (ii) *global* queries, which require information about most or all vertices (e.g. graph diameter). Orthogonally to the amount of vertices, a query may require access to (i) a single snapshot, or (ii) to a range of snapshots in the sequence. The combinations of these two dimensions yield four main types of queries over evolving graphs, all of which need to be supported. An example of a local query over a single snapshot is *"What was the 2-hop neighborhood of vertex A at time instance t?"*, whereas an example of a global query over the entire graph history is *"Find the average number of friends of each member of a social network in each month since the network's creation."*

The key distinctive feature of our approach, called HiNode, is that it moves away from the concept of using deltas to reconstruct specific snapshots, e.g., as in [29,21,19,20]. In particular, as explained in [20], a system may be organized in a *time-centric* manner (i.e. the data is indexed according to the time instances), or in an *entity-centric* approach (i.e. the data is indexed according to vertices and edges and their individual history). In this paper, we propose the first purely *entity-centric*, and more specifically, *vertex-centric* approach to organizing evolving graphs.

In our approach, each graph vertex is associated with its history in the evolving graph defined as the sequence of graph snapshots in which the vertex exists. This history is stored within the vertex, and is structured in such a way so that various queries can be supported. In particular, we model the history of vertices as time intervals where various geometric operations, such as stabbing queries[1], are supported. These geometric operations implement a fundamental set of access operations on the snapshots of the graph sequence. This local storage of history offers efficient handling of local queries by allowing the effective reconstruction of simply the sub-graph of interest instead of the complete graph. Furthermore, the time-interval representation allows the efficient support of queries involving a range of snapshots.

An additional feature of our approach is that it is very flexible with respect to the supported notion of time. In [19] the authors differentiate between the two notions of time used by researchers in the literature. More specifically, *transaction time* represents the time that an event takes place (i.e. the moment that an object is stored or deleted from a database) whereas *valid time* signifies the time period in which an object was valid (i.e. the time interval that an object existed in a database). In the transaction time setting updates can only occur in an append-like manner (i.e. an update in a field changes the value of the most recently stored value) whereas in the valid time setting updates can refer to any previously stored value (i.e. an update may change any previously stored interval retroactively). Transaction time can be emulated in the valid

[1] Given a query point $p \in \mathcal{R}$ and a set of N intervals on the real line, a stabbing query returns all intervals that overlap p.

time setting by restricting updates to intervals that begin on the time moment of the update. Our approach not only supports either transaction or valid time but can be further generalized with minor modifications to support parallel universe time.

Additionally, we experimentally show that our technique is also efficient for global queries as well, especially when a range of snapshots is considered, because the overheads of a complete graph reconstruction are outweighed by the benefits of accessing less data. Finally, our framework can be parallelized in a straightforward manner with the only realistic assumption being that a single vertex with its history fits in memory.

Our principal goal is to show that HiNode, as a pure vertex-centric storage model, exhibits great potential. This is showcased both theoretically through a qualitative system comparison with our two main competitors (G^* [23,34] and TGI [20]) as well as experimentally using the G^* parallel graph processing system which is available for download. To accomplish this goal, we focused on the space usage of the storage model as well as its functionality. Historically, space efficiency was mainly considered due to high memory prices. Nowadays, this is not a severe constraint anymore unless the data is really massive (for example, consider the social network of Facebook and all its history). One of the reasons for which we still thrive for space efficiency is precisely the existence of massive data sets. Another reason is the shifting of computation to smaller devices that do not have ample memory. Additionally, having a better space utilization results in reducing access to slower storage and thus gaining running speed. One of our goals is to show that in this pure vertex-centric approach the space overhead is minimal leading to indirect time savings in queries as well.

In addition, HiNode can support various sets of queries focusing largely on local queries where the pure vertex-centric approach is naturally suitable and shows great potential. This is indicated by theoretical as well as experimental results. Finally, we provide experimental and theoretical indications that HiNode can support general queries very well without going into great depths. This is because, although this paper provides ample evidence that HiNode is competitive, a full implementation of a historical graph database requires substantially more modules that unfortunately due to the different architecture of the G^* system cannot be implemented efficiently. We also note here that we were not able to compare experimentally TGI with HiNode since it follows a different architecture than the G^* system and incorporating it in the existing system would be considerably impractical.

In summary, we make the following contributions:

- We introduce the first purely *vertex-centric* approach to the organization of evolving graphs. Our approach is especially suitable for local queries that involve one or more graph snapshots.
- We show that our storing and indexing approach is more space efficient than existing approaches.

- We present a set of primitive operators on top of which many complex historic graph execution plans can be built. We provide evidence that these operations can be efficiently supported with results comparable or even better than those of other time-centric approaches.
- Our approach can efficiently support more general notions of time with respect to the evolution of the graph. The evolution of the graph may be represented in a linear or a tree-like fashion whereas all solutions up to this point support a linear notion of time.
- Our approach handles in a very natural way incremental updates that generate new snapshots. The granularity of evolutions can be arbitrarily fine whereas all other solutions try to overcome this problem by either employing logging or by grouping together many updates which naturally leads to handling a rather small number of snapshots.
- To prove the practicality of our approach, we have implemented and incorporated it in the G^* parallel graph processing system [23,34]. We experimentally show that, due to the low cost of accessing stored data, the reconstruction overheads are outweighed in most of the cases, and our solution is efficient even for queries that require the reconstruction of complete snapshots.

The rest of this work is organized as follows. In the next section we discuss previous work. In Section 3, we present the main definitions and notation. We introduce our storage model in Section 4. We present core algorithms and practical considerations in Section 5. In Section 6, we conduct experimental evaluation of our method. In Section 7 we discuss multiple universes and efficiently registering updates and we conclude in Section 8.

2 Previous Work

There have been two main research directions over the previous years with regards to graph processing. The first approach includes systems such as Trinity [33], Pregel [25], Giraph [14] and others (e.g. GBase [17], Pegasus [18], Cassovary [5], distributed graph management systems [27]) that focus on single snapshot processing. The main characteristic of these systems is that they operate on single very large graphs, as opposed to a collection of related graph snapshots. The second category is concerned with handling evolving sequences of large graphs that mostly resemble the history of a network. The following paragraphs analyze the work conducted in this research direction, while a comprehensive survey can be found in [22].

Perhaps, the closest proposals to ours are those in [19,23,34]. In general, these techniques rely on storage of snapshots and deltas, which exhibits a trade-off between space and time. Having a large amount of snapshots results in deltas of small size but the space cost is substantial since we need to maintain many copies of the graph. On the other hand, having a handful of snapshots means that deltas will be quite large. By contrast, we follow a purely vertex-

based approach without storing explicitly snapshots and deltas, and we prove our asymptotic space-optimality.

In [19] the authors propose a solution based on the notion of valid time, thus making the overall problem more challenging. Their proposed system is composed of two parts. The first component is a disk-stored tree-like index structure called DeltaGraph that contains specific materialized snapshots and differential functions, while the second component is an in-memory structure that stores a number of materialized snapshots. An extension to DeltaGraph geared towards vertex-centric queries, called Temporal Graph Index (TGI), was proposed in [20]. Similarly to the techniques in [19,20] we also support the notion of valid time and we employ non-trivial data structures, as explained later. However, our proposal is more space-efficient than [19,20].

A parallel graph processing system named G^* has been proposed in [23,34]. G^* stores each vertex along with its history in a server only once, regardless of the number of snapshots in the sequence it exists in. This minimizes the overall space used for storing changes since duplicate data is avoided. However, the system indexes both the vertices as well as the historical information within the vertices which may lead to a quadratic blowup in space for pointer data (but not for the graph data). For that reason the authors focus on a small number of snapshots and employ logging in order to store all intermediate history between successive snapshots. In addition, because of the extra pointer data G^* does not follow a pure vertex-centric approach. In the evaluation results, we show that we can improve on [23,34] by up to an order of magnitude.

An earlier attempt to efficiently handle historical graph queries has appeared in [29], which is later generalized in [21]. Evolving graph sequences can also be engineered to permit efficient evaluation of specific features or queries, such as historical reachability queries [32], mining the most frequently changing component [35], continuous pattern detection [12] or shortest path distance queries [16]. Finally, in [6,9], space-efficient methods for compressing graph sequences are proposed. However, these techniques are offline, since they require the entire sequence to be given in advance. For operating online sequences, the authors express some thoughts without providing concrete implementation details. Furthermore, they only consider graphs with no additional attributes on their vertices and edges (e.g. weights). In general, the use of compression in our techniques is an interesting direction for future work.

3 Preliminaries

We follow a linear notion of time and define it to be a strictly increasing quantity measured in indivisible time intervals. We discuss a tree-like multiple history in Section 7. Let $\mathcal{G} = \{G_1, G_2, G_3, \ldots\}$ be the sequence of graph snapshots to be stored and accessed. The sequence does not have a final snapshot, in the sense that it is constantly evolving. For a graph $G_i \in \mathcal{G}$, the snapshot $G_i = (V_i, E_i)$ corresponds to the graph G at time instance i and is characterized by a set of vertices V_i and a set of incoming and outgoing edges E_i,

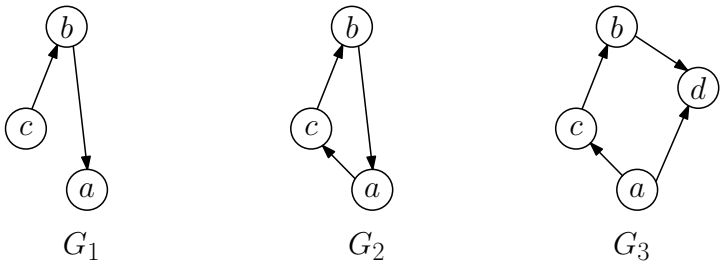

Fig. 1 Evolving Graph Sequence. G_2 is obtained by inserting an edge to G_1, while G_3 is obtained by creating a vertex, inserting two edges and deleting an edge from G_2 respectively

with each vertex or edge possessing its own set of (possibly multi-valued) attributes. Conceptually, one may obtain the snapshot of G_i by applying a set of insertions, deletions or value updates to the vertices and edges of G_{i-1}. Moreover, since our theoretical model works with valid time, a user is able to update a specific snapshot even though it may not currently be the last in the sequence (i.e., a user may update G_{i-j}, where $i > j$, even though $\mathcal{G}$ currently holds i snapshots).

The sequence $\mathcal{G}$ can also be defined with respect to its vertices and edges ($\mathcal{V}$ and $\mathcal{E}$ respectively). Therefore, it follows naturally that $\mathcal{V} = V_1 \cup V_2 \cup V_3 \cup \ldots$ and $\mathcal{E} = E_1 \cup E_2 \cup E_3 \cup \ldots$. We use m to denote the total count of changes (insertions, deletions and value updates) made throughout the sequence. Note that since new snapshots may be added to the sequence, the value of m becomes larger with each new snapshot. As a direct consequence of the above, we can deduce that at any time instance, the total count of snapshots, vertices or edges is up to m, i.e. $|\mathcal{G}| \leq m$ and $|\mathcal{V}| + |\mathcal{E}| \leq m$. An example of an evolving graph sequence is depicted in Figure 1. The notation used throughout the remainder of this work is summarised in Table 1. The algorithms and data structures of this work are expressed in the standard two-level external memory model proposed by Aggarwal and Vitter [1].

4 Storage Model

We propose a storage model for the graph sequence $\mathcal{G}$. This storage model supports a variety of fundamental access and update operations that allow the user to access any graph G_i (either the whole graph or a particular subgraph) as well as alter the graphs at any time instance. Furthermore, the storage model does not rely on there being no changes in the schema of the vertices and supports different attributes in the same vertex over time. In the following, we begin with an overview of the proposed data structure, which comprises nested elements at different levels, followed by a description of the supported operations. We conclude the section by providing an asymptotic analysis of the space and time complexities.

Table 1 Notation Table

Symbol	Description
G_i	A snapshot of graph G at time instance i
V_i	The set of vertices of G_i
E_i	The set of incoming and outgoing edges of G_i
$\mathcal{G}$	A sequence of G_i for various i
$\mathcal{V}$	The set of all vertices in $\mathcal{G}$
$\mathcal{E}$	The set of all edges in $\mathcal{G}$
$\|v\|$	Size of vertex v, i.e. the count of attributes and edges of v across all of $\mathcal{G}$
$\|v_t\|$	Size of vertex v at time t
$\mathcal{S}$	Total size of all vertices in V_i
m	The total count of changes (insertions, deletions, updates) made from the first snapshot to the (currently) last snapshot in $\mathcal{G}$
$\mathcal{I}$	External interval tree that maintains the "lifetime" of each vertex in $\mathcal{V}$
$\mathcal{T}^v_{t_s,t_e}$	An interval in $\mathcal{I}$ signifying that vertex v in $\mathcal{G}$ is active between the time instances t_s and t_e
$\mathcal{D}_v$	Diachronic node of vertex v
f	An identifier of a particular attribute (e.g. name, weight etc.) or edge
$\mathcal{I}_v$	External interval tree of $\mathcal{D}_v$ that maintains information regarding all the attributes of v
A^f_v	B-tree for the attribute with identifier f of vertex v
$\mathcal{B}_v$	A B-tree used as an index over all A^f_v trees
$\mathcal{B}$	A B-tree used as an index over the identifiers of all the diachronic nodes
B	Disk block size

4.1 Data Structure Overview

In this section, we will provide an overview of the data structure and we will use a data structure built on the evolving graph sequence of Figure 1 as a running example (Figure 3). Recall that $\mathcal{G} = \{G_1, G_2, G_3, \ldots\}$ is a sequence of graph snapshots with each $G_i \in \mathcal{G}$ corresponding to a snapshot of the graph G at time instance i. A vertex $v \in G_i$ is characterized by a set of attributes (e.g. color), a set of incoming edges from the other vertices of G_i and a set of outgoing edges to the other vertices of G_i. We construct an external interval tree $\mathcal{I}$ that maintains a set of intervals $\{\mathcal{T}^v_{t_s,t_e}\}$ where an interval $\mathcal{T}^v_{t_s,t_e}$ signifies the "lifetime" of a vertex v, i.e. from time instance t_s to time instance t_e. We mark a vertex to be "active" (alive) up until the latest time instance, by setting the t_e value to be $+\infty$. As an example, in Figure 3 we can see that at time instance 3, interval tree $\mathcal{I}$ stores four intervals, namely $\{\mathcal{T}^a_{1,\infty}, \mathcal{T}^b_{1,\infty}, \mathcal{T}^c_{1,\infty}, \mathcal{T}^d_{3,\infty}\}$. Finally, each interval $\mathcal{T}^v_{t_s,t_e}$ is augmented with a pointer (handle) to an additional data structure for each vertex v, called *diachronic node*.

A diachronic node $\mathcal{D}_v$ of a vertex v maintains a collection of data structures corresponding to the full vertex history in the sequence $\mathcal{G}$, i.e. when that vertex was inserted, all corresponding changes to its edges or attributes and finally its deletion time (if applicable). More formally, a diachronic node $\mathcal{D}_v$ maintains an external interval tree $\mathcal{I}_v$ which stores information regarding all

of v's characteristics (attributes and edges) throughout the entire $\mathcal{G}$ sequence. An interval in $\mathcal{I}_v$ is stored as a quadruple $(f, \{\ell_1, \ell_2, \ldots\}, t_s, t_e)$, where f is the identifier of the attribute that has values ℓ_1, ℓ_2, ... during the time interval $[t_s, t_e]$. Note that an edge belonging to v (i.e. one endpoint of the edge is v), can be represented as an attribute of v by using one value ℓ_i to denote the other end of the edge, another value ℓ_j to mark the edge as incoming or outgoing and a last value ℓ_h that is used as a handle to the diachronic node corresponding to the vertex in the other end of the edge. The remaining ℓ values can be used to store the attributes of the edge themselves (e.g. weight). In the running example of Figure 3, we can see that at time instance 3, the interval tree $\mathcal{I}_a$ of the diachronic node $\mathcal{D}_a$ stores four intervals: one for the label of the vertex and three for all the edges that have occurred thus far for vertex a.

Additionally, the diachronic node maintains a B-Tree for each attribute and for each individual edge of the vertex. The B-tree corresponding to the attribute with identifier f of vertex v is denoted as A_v^f and is used to maintain the entire history of changes of f between the different snapshots in $\mathcal{G}$. Each record in A_v^f is a triple $(\{\ell_1, \ell_2, \ldots\}, t_s, t_e)$ where ℓ_1, ℓ_2, ... are the values of f during the time interval $[t_s, t_e]$. The edges of v are represented in a similar manner to attributes taking into account the values ℓ_i, ℓ_j and ℓ_h which were discussed on the previous paragraph. Using the A_v^f trees, which are built on the $[t_s, t_e]$ intervals, we can support stabbing queries for a particular set of attributes within the diachronic node, without aggravating asymptotically the space usage. Given the fact that the count of A_v^f trees is dependant on the edge count of v and to facilitate efficient searching of a specific A_v^f tree, we maintain a B-Tree $\mathcal{B}_v$ over all A_v^f trees. Finally, we maintain the location of all diachronic nodes using a B-Tree dictionary $\mathcal{B}$ built on the IDs of the diachronic nodes. With regard to the example of Figure 3, at time instance 3 the diachronic node of vertex a has four A_v^f B-trees each containing one record.

Figure 2 shows the proposed data structure, where $|v|$ is the size of v (i.e. the count of attributes and edges of v across all of $\mathcal{G}$). The full arrows are handles to diachronic nodes from the intervals in $\mathcal{I}$ while the dashed arrows signify handles to diachronic nodes from $\mathcal{B}$. Depending on the operation we may use either option to locate a specific diachronic node. The full example of the data structure built on the evolving graph sequence of Figure 1 can be seen in Figure 3. We name our storage model as *HiNode* standing for "History in the (diachronic) Node".

4.2 Basic Operations

We implement a basic set of operations over the graph sequence that can be used as primitive for implementing more complex operations concerning a subgraph (or the entire graph) at a particular time instance. Henceforth, we use the terms fields and attributes interchangeably to refer to the attributes

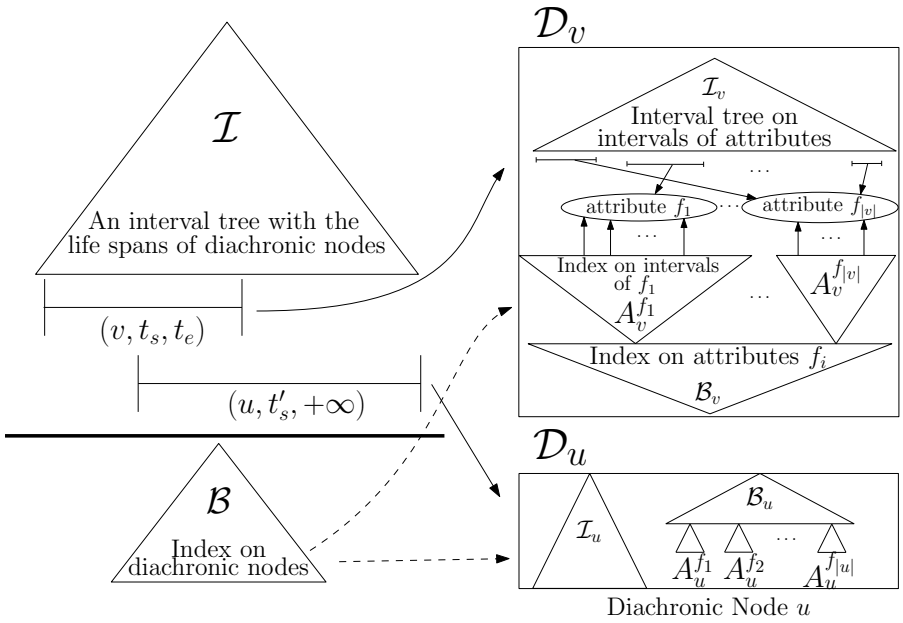

Fig. 2 A bird's-eye view for an instance of our proposed data structure.

of each vertex and assume that all operations refer to a vertex in the sequence labeled as v. Each operation description is accompanied by pseudocode.

`InsertVertex` (Algorithm 1) creates an interval $\mathcal{T}^v_{t_s,t_e}$ that corresponds to a new vertex v in the sequence $\mathcal{G}$ that is inserted at time t_s. Furthermore, an empty diachronic node $\mathcal{D}_v$ is created for v and a pointer p_v to $\mathcal{D}_v$ is attached to $\mathcal{T}^v_{t_s,t_e}$ and inserted in $\mathcal{B}$. Finally, the interval $\mathcal{T}^v_{t_s,t_e}$ is inserted in $\mathcal{I}$ and the p_v pointer is returned. If at the time of insertion, the end time t_e is not known, we set it to infinity. Note that even if v gets deleted at some time instance t, we keep $\mathcal{D}_v$ to facilitate historical queries.

`ReadAttribute` (Algorithm 2) returns the set of `values` $\{\ell_1, \ell_2, \ldots\}$ of the `field` f in vertex v at time t. To realize this operation we retrieve the diachronic node $\mathcal{D}_v$ by querying $\mathcal{B}$ for `id` v. Afterwards, we obtain the A_v^f tree using $\mathcal{B}_v$ and perform a query for time t. If there exists a set of values $S = \{\ell_1, \ell_2, \ldots\}$ for f at the time instance t it is returned, otherwise the operation returns a `NULL` value.

`WriteAttribute` (Algorithm 3) assigns a set of `values` $S = \{\ell_1, \ell_2, \ldots\}$ which are valid for the time interval $[t_s, t_e]$ to the `field` f of vertex v. To achieve this, firstly we obtain the diachronic node $\mathcal{D}_v$ of v by searching $\mathcal{B}$ for `id` v. Following that, we perform a stabbing query on the respective A_v^f tree of f for each of the t_s and t_e endpoints. If the stabbing queries do not return any interval then the `field` f does not have any values associated with it in the time interval $[t_s, t_e]$ and we insert the relevant data in $\mathcal{I}_v$ and A_v^f directly; otherwise, we retrieve the (at most two) returned interval(s) (e.g. $[t'_s, t'_e]$ and

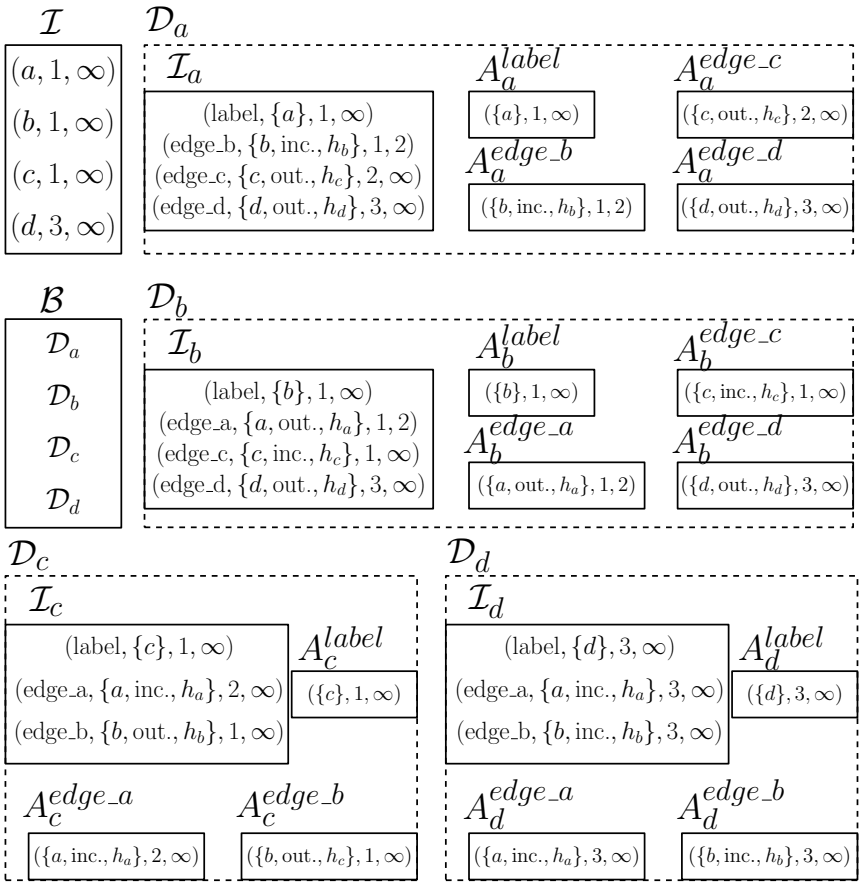

Fig. 3 The data structure built on the graph sequence of Figure 1 as seen at time instance 3.

Algorithm 1 InsertVertex(v, t_s, t_e)

Input: vertex id v, start time t_s, end time t_e
Output: a pointer p_v to the diachronic node of v

1: $\mathcal{T}^v_{t_s,t_e} \leftarrow$ **new** interval (v, t_s, t_e)
2: $p_v \leftarrow$ pointer(**new** diachronic node(v))
3: $\mathcal{T}^v_{t_s,t_e}$.attach(p_v)
4: $\mathcal{B}$.insert(p_v) ▷ p_v is inserted in $\mathcal{B}$ along with the ID of v
5: $\mathcal{I}$.insert($\mathcal{T}^v_{t_s,t_e}$)
6: **return** p_v

$[t''_s, t''_e]$), remove their portion that is being overlapped by $[t_s, t_e]$ and finally insert the relevant data in $\mathcal{I}_v$ and A^f_v. The two cases are described in detail in Appendix A.

Finally, the `WriteAttribute` operation is also used in a similar manner to delete a particular interval by specifying the field f and the interval $[t_s, t_e]$ to

Algorithm 2 ReadAttribute(v, f, t)

Input: vertex id v, attribute or field f, time instance t
Output: a set of values $S = \{\ell_1, \ell_2, \ldots\}$ corresponding to the values of f at time t; `NULL` otherwise

1: $\mathcal{D}_v \leftarrow \mathcal{B}$.query($v$)
2: $A_v^f \leftarrow \mathcal{D}_v.\mathcal{B}_v$.query($f$)
3: $S \leftarrow A_v^f$.stab(t)
4: **if** $S \neq \emptyset$ **then return** S
5: **else return** `NULL`
6: **end if**

Algorithm 3 WriteAttribute(v, f, $\{\ell_1, \ell_2, \ldots\}$, t_s, t_e)

Input: vertex id v, attribute or field f, a set of values $\{\ell_1, \ell_2, \ldots\}$, start time t_s, end time t_e
Output: -

1: $\mathcal{D}_v \leftarrow \mathcal{B}$.query($v$)
2: $A_v^f \leftarrow \mathcal{D}_v.\mathcal{B}_v$.query($f$)
3: $hasValues \leftarrow false$ ▷ Boolean used to distinguish between the two cases
4: **if** A_v^f.stab(t_s)$\neq \emptyset$ **or** A_v^f.stab(t_e)$\neq \emptyset$ **then**
5: Retrieve the interval(s) from A_v^f
6: $hasValues \leftarrow true$
7: **end if**
8: **if** $\neg hasValues$ **then** ▷ Case 1
9: $\mathcal{I}_v$.insert($(f, \{\ell_1, \ell_2, \ldots\}, t_s, t_e)$)
10: A_v^f.insert($(\{\ell_1, \ell_2, \ldots\}, t_s, t_e)$)
11: **else** ▷ Case 2
12: Using the interval(s) retrieved from A_v^f, retrieve the corresponding interval(s) in $\mathcal{I}_v$
13: **if** $\nexists[t'_s, t''_e]$ **then**
14: **if** $t'_s < t_s < t'_e < t_e$ **then** ▷ Subcase a
15: In $\mathcal{I}_v$, A_v^f: Delete $[t'_s, t'_e]$. Insert $[t'_s, t_s)$, $[t_s, t'_e)$, $[t'_e, t_e]$
16: **else if** $t_s < t'_s < t_e < t'_e$ **then** ▷ Subcase b
17: In $\mathcal{I}_v$, A_v^f: Delete $[t'_s, t'_e]$. Insert $[t_s, t'_s)$, $[t'_s, t_e)$, $[t_e, t'_e]$
18: **else if** $t'_s < t_s < t_e < t'_e$ **then** ▷ Subcase c
19: In $\mathcal{I}_v$, A_v^f: Delete $[t'_s, t'_e]$. Insert $[t'_s, t_s)$, $[t_s, t_e)$, $[t_e, t'_e]$
20: **end if**
21: **else** ▷ Subcase d
22: In $\mathcal{I}_v$, A_v^f: Delete $[t'_s, t'_e]$, $[t''_s, t''_e]$. Insert $[t'_s, t_s)$, $[t_s, t'_e)$, $[t''_s, t_e)$, $[t_e, t''_e]$
23: **end if**
24: **end if**

be deleted and passing a `NULL` value as the set of values. Note that the deletion operation is meaningless in the transaction time setting.

We conclude the operation by pointing out that the first case represents the insertion of data corresponding to a particular vertex's field, while the second case can be seen as *correcting* the data that the vertex already had. To that end, we do not permit cases where the insertion of an interval would delete an already existing interval (e.g. $t_s < t'_s < t'_e < t_e$) as that would result in the loss of information. In those cases, the user should explicitly delete the related intervals before inserting the new one.

Algorithm 4 ReadVertex(v, t)

Input: vertex id v, time instance t
Output: a pointer p_v to an object u that corresponds to v as seen at t
1: $\mathcal{D}_v \leftarrow \mathcal{B}$.query($v$)
2: $\mathcal{P} \leftarrow \mathcal{D}_v.\mathcal{I}_v$.stab($t$)
3: $u \leftarrow$ **new** vertex(v,t)
4: **for each** $f = (attribute$ **or** $edge)$ **in** $\mathcal{P}$ **do**
5: u.add(f)
6: **end for**
7: **return** pointer(u)

`ReadVertex` (Algorithm 4) returns a pointer to an object u that corresponds to the vertex v as seen at time t. This is realized by obtaining the diachronic node $\mathcal{D}_v$ of v using $\mathcal{B}$ and then performing a stabbing query to $\mathcal{I}_v$. The stabbing query returns the set of values at time instance t for each field f in v. After following this approach, all the resulting objects are collected and put in an object u which is returned by p_u.

4.3 Analysis of Space and Time Complexities

Our space and time cost analysis is based on the relevant costs of B-Trees and external interval trees. Assuming that the first snapshot in the sequence is empty, each of the m changes occurring in the time-evolving sequence is ultimately stored $O(1)$ times in $O(1)$ linear-sized data structures. More specifically, in the `Insert Vertex` operation we insert an interval in $\mathcal{I}$ and a record in $\mathcal{B}$ for each newly created vertex while in each `WriteAttribute` operation we insert up to three intervals and records in $\mathcal{I}_v$ and the corresponding A_v^f B-tree respectively. This brings the total space usage for m changes to $O(m/B)$ which is asymptotically optimal with respect to the number of changes.

We analyze the time cost of each operation. The `InsertVertex` operation requires $O(\log_B m)$ time, since the insertion of $\mathcal{T}^v_{t_s,t_e}$ in $\mathcal{I}$ and the insertion of the diachronic node pointer in $\mathcal{B}$ each require $O(\log_B m)$ time (the creation of the diachronic node itself requires constant time).

The `ReadAttribute` operation requires $O(\log_B m)$ time in total. Retrieving $\mathcal{D}_v$ by querying $\mathcal{B}$ and obtaining A_v^f require $O(\log_B m)$ time each. Finally, the query in A_v^f also requires $O(\log_B m)$ time.

We will analyze the `WriteAttribute` operation by separately analyzing its two cases. Firstly, obtaining $\mathcal{D}_v$ is done in $O(\log_B m)$ time. To determine which case stands true, we perform two calls to `ReadAttribute` that require $O(\log_B m)$ time in total. In the first case we perform two insertions, each requiring $O(\log_B m)$ time.

In the second case, searching for $[t'_s, t'_e]$ (and potentially $[t''_s, t''_e]$) in $\mathcal{I}_v$ can be done in $O(\log_B m)$ total time. In any of the resulting subcases we perform a series of $O(1)$ deletions and insertions each requiring $O(\log_B m)$ time. Thus, the second case also requires $O(\log_B m)$ time, yielding $O(\log_B m)$ total time for the operation.

To access a full vertex v at time t, we retrieve $\mathcal{D}_v$ by querying $\mathcal{B}$ and then we perform a stabbing query to $\mathcal{I}_v$ by using the operation `ReadVertex`. The total cost is $O(\log_B m + |v_t|/B)$, where $|v_t|$ is the size of vertex v at time t.

The following theorem summarizes the obtained space and time results.

Theorem 1 *We can maintain a time-evolving graph sequence in a data structure using $O(m/B)$ space, where m is the total number of changes in the sequence and B is the disk block size. The data structure supports the following basic operations:*
1) `InsertVertex` in $O(\log_B m)$ time,
2) `ReadAttribute` in $O(\log_B m)$ time,
3) `WriteAttribute` in $O(\log_B m)$ time,
4) `ReadVertex` in $O(\log_B m + |v_t|/B)$ time, where $|v_t|$ is the size of the vertex v at time t (i.e. the count of attributes and edges of v at time t).

4.4 Qualitative Comparison

Here we provide a qualitative comparison between different methods. To do that, we assume only transaction time, since not all solutions support valid time, as we and TGI do. This comparison is based on the formal Δ framework introduced in [20], which is summarized as follows. An *ephemeral vertex* is a vertex at a specific time instance. Thus, one needs to specify the time instance t for which we are interested to retrieve the ephemeral vertex. The ephemeral vertex contains an identifier, a list of incoming and outgoing edges (a list of edges in the case of an undirected graph) as well as a set of attributes that are attached to this particular vertex or to an edge. An ephemeral edge is similarly defined. A Δ is a set of ephemeral vertices and edges at potentially various times instances. All kinds of graph operations can be defined on Δs in order to compress and make more efficient queries on time instances or time intervals. An *event* is the minimum change that registers a new instance of the graph. As a Δ, an event is simply the set of ephemeral vertices and edges that constitute the changes between two successive time instances. An *eventlist* is a sequence of successive events sorted in a chronological order. An eventlist is specified by the time interval $[t_{start}, t_{end}]$ of the respective Δs. Finally, a *snapshot* at a particular time instance t is the set of all ephemeral vertices and edges at time instance t.

Following the above definitions, we provide a qualitative comparison between different methods in Table 2. Column "Space" refers to the space required by each model while the rest of the columns refer to the access time required by each respective operation.

Ephemeral operations correspond to accessing a particular specified object (graph, subgraph or vertex) at a particular time instance while *Versioned* operations correspond to accessing a particular object in a time interval. The 1-Hop operation corresponds to accessing all adjacent vertices of a particular

Table 2 The table is split into two pieces in order to facilitate readability. Comparison of storage models w.r.t. space and access time on various operations. Multiplicative and additive constant factors are discarded. Some of the following parameters either correspond to the actual size of an object (e.g., size of a vertex in the operation Ephemeral Vertex) or to a mean value (e.g., mean size of a vertex in operation Versioned Subgraph). $m \rightarrow$ total number of stored changes; $|S| \rightarrow$ total size of snapshot; $|E| \rightarrow$ eventlist size; $h \rightarrow$ height of a tree of snapshots used in TGI. It is upper bounded by $\log m$; $|C| \rightarrow$ number of changes within a vertex. It is upper bounded by m; $d \rightarrow$ the degree of a vertex; $|W| \rightarrow$ size of an ephemeral subgraph W; $|A| \rightarrow$ size of an ephemeral vertex.

Model	Space	Edge Insertion	Snapshot	Ephemeral Vertex	Versioned Vertex
OPTIMAL	m	1	$\|S\|$	$\|A\|$	$\|C\|$
LOG	m	1	m	m	m
COPY	m^2	m	$\|S\|$	$\|S\|$	$\|S\|\|C\|$
COPY+LOG	$\frac{m^2}{\|E\|}$	$\frac{m}{\|E\|}$	$\|S\| + \|E\|$	$\|S\| + \|E\|$	m
TGI	hm	hm	$h\|S\| + \|E\|$	$h\|S\| + \|E\|$	$\|C\|\|S\|$
G^*	$m + \frac{m^2}{\|E\|\|C\|}$	$\lg m + \|E\| + \frac{m}{\|E\|}$	$\|C\|\frac{m}{\|E\|} + \|E\|$ $+ \lg m + \|S\|$	$\|C\|\frac{m}{\|E\|} + \|E\|$ $+ \lg m + \|A\|$	$\|C\|\left(\frac{m}{\|E\|} + \lg m\right) + \|E\|$
HiNode	m	$\lg m + \lg \|C\|$	$\|S\| \lg \|C\|$	$\lg m + \|A\| + \lg \|C\|$	$\lg m + \|C\| + \lg \|C\|$

Model	Ephemeral 1-Hop	Versioned 1-Hop	Ephemeral Subgraph	Versioned Subgraph
OPTIMAL	d	$d\|C\|$	$\|W\|$	$min\{\|W\|\|C\|, m\}$
LOG	m	m	m	m
COPY	$\|S\|$	$\|S\|\|C\|$	$\|S\|$	$\|S\|\|C\|$
COPY+LOG	$\|S\| + \|E\|$	m	$\|S\| + \|E\|$	m
TGI	$h\|S\| + \|E\|$	$\|C\|\|S\|$	$h\|S\| + \|E\|$	$h\|S\| + \|C\|\|W\|$
G^*	$\|C\|\frac{m}{\|E\|} + \|E\|$ $+ \lg m + \|A\| + d$	$\|C\|\left(\frac{m}{\|E\|} + \lg m\right)$ $+ \|E\| + d\|C\|$	$\|C\|\frac{m}{\|E\|} + \|E\|$ $+ \lg m + \|W\|$	$\|C\|\left(\frac{m}{\|E\|} + \lg m\right)$ $+ \|E\| + \|C\|\|W\|$
HiNode	$\lg m + \|A\| + \lg \|C\| + d$	$\lg m + d\|C\| + \lg \|C\|$	$\|W\| \lg \|C\|$	$\|C\|\|W\| \lg \|C\|$

vertex. The OPTIMAL row corresponds to the ideal space and time access costs for the operations in the worst-case.[2]

The LOG method corresponds to a single initial snapshot with an eventlist that stores all changes. The COPY method corresponds to a snapshot for each change without eventlists. COPY corresponds to storing each snapshot as a separate graph in a graph database system. One could combine these methods (COPY+LOG) by allowing a sequence of snapshots with eventlists that record the changes between them. In the following paragraphs, we describe the space cost required by the TGI and G^* approaches. Furthermore, we outline the time cost for the operations in the first half of Table 2 (the rest follow the same logic and are defined similarly).

TGI is based on the COPY+LOG idea which, however, is considerably tuned so that it allows a hierarchical structure of snapshots combined with partitioning of eventlists into small chunks in order to achieve better locality. Furthermore, the system supports lists of different instances of diachronic

[2] For G^*, TGI and our solution (and to a lesser extent for the other methods), one could indeed describe the complexity w.r.t. a variety of parameters and provide a more detailed description. However, doing so would certainly not permit the direct comparison between the methods and would thus invalidate the very reason for which this table is provided.

vertices within the snapshots ("vertex chains") to facilitate vertex-centric operations. In addition, one of its merits is the dynamic partitioning of the graph. We choose not to incorporate it in this comparison for reasons of uniformity since all other solutions do not support such an explicit partitioning process but consider it as an additional external mechanism. Finally, updates in TGI are only supported in batch mode and the system does not allow for online small changes.

Before outlining the space and time complexity attained by TGI, we note that the parameter h corresponds to the height of the hierarchical tree structure employed by the TGI approach. On each level of the hierarchical structure, TGI stores the delta difference between each parent node from its child, thus bringing the total space cost to $O(hm)$. To make an edge insertion we have to materialize the hierarchy of differences which results in a cost of $O(hm)$. Obtaining a snapshot in TGI is accomplished by traversing a root-to-leaf path in the tree structure and materializing any delta associated to the snapshot on each node. This results in a $O(h|S| + |E|)$ cost with the $|E|$ factor corresponding to any remaining changes in the leaf-eventlist stored in the leaf that must be applied to obtain the snapshot. The Ephemeral Vertex operation has the same cost since it is performed similarly to a snapshot retrieval with the difference being that the system only retrieves the subset of deltas associated to the queried node. Finally, in the Versioned Vertex operation, the system makes use of the vertex chains to obtain the eventlists corresponding to each change in the node in a given query interval, resulting in an $O(|C||S|)$ cost for the $|C|$ eventlists that need to be filtered for the queried node.

G^* is a combination of a vertex-centric approach and the COPY+LOG method, where differences are stored in a compact way by employing indirection. The system achieves that by employing an index called Compact Graph Index (CGI). CGI manages collections of vertex version locations on disk (VL maps), with each vertex version being represented once for each combination of graphs that it exists in (see Section 6.1 for more practical details). This means that, for successive snapshots, only differences are stored. As a result, this fact complicates logging of operations between snapshots but to a small degree (the authors do not consider it, since their focus is only on the storage of a rather small number of successive snapshots).

G^* stores each of the m changes only once and employs CGI to index the different versions that occur. The number of VL maps increases quadratically with the number of graphs [23] and since G^* employs logging to store all intermediate history between successive snapshots, the space cost of G^* is $O(m+\frac{m^2}{|E||C|})$. The edge insertion cost of G^* is $O(\lg m+|E|+\frac{m}{|E|})$ since it follows a similar approach to the COPY+LOG method with the addition of traversing the index and applying any logged updates. In order to retrieve a snapshot, G^* iterates through the VL maps and retrieves all vertices that are related to the queried time instance. It then applies any logged updates before returning the snapshot resulting in a total cost of $O(|C|\frac{m}{|E|} + |E| + \lg m + |S|)$. The same procedure is followed for the Ephemeral Vertex operation but instead of

returning all vertices for the queried time instance, the system limits its query to the given vertex. Finally, in the Versioned Vertex operation G^* employs the same procedure but may require visiting multiple paths in the index since the query refers to a time interval instead of a particular time instance.

We provide an outline of the complexities attained by HiNode. The space cost of HiNode is linear to the number of changes (Theorem 1). The time complexity to insert an edge is the same as the one for writing an attribute (see the WriteAttribute operation in Theorem 1). First, we obtain the relevant diachronic node by traversing a B-Tree in logarithmic time and then we insert the edge. However, a more accurate description of the complexity can be attained by noting that the number of changes in the node is $|C|$ and, since $|C|$ is bounded by m, we obtain the final result. To obtain a snapshot, we perform multiple ReadVertex operations on all vertices that are "alive" at that particular time instance. The cost of retrieving a single vertex is $O(log|C|)$, which refines the $O(\log_B m)$ definition in Theorem 1. Since, the total size of the snapshot is $|S|$, the total cost is $|S|log|C|$. In order to read an ephemeral vertex we need to locate its diachronic node using the B-Tree ($\log m$), retrieve the vertex at that particular time instance ($\log |C|$) and read its attributes and edges ($|A|$). Reading a versioned vertex is performed in a similar manner but, instead of the factor $|A|$, the operation costs $|C|$ since we need to retrieve all the relevant changes in query interval. The ephemeral 1-hop operation corresponds to retrieving an ephemeral vertex plus accessing its d neighbors. In the versioned 1-hop operation, there exist up to $d * |C|$ different neighbors for the node throughout the query interval so the cost is equal to retrieving the vertex and accessing all of its potential neighbors. The ephemeral subgraph operation follows the same logic as the snapshot operation with the difference being that instead of $|S|$ the total cost depends on the total size $|W|$ of the ephemeral subgraph (attributes and edges of all nodes in the subgraph) and is $|W| \log |C|$. Finally, in the versioned subgraph operation the total cost is $|C||W| \log |C|$ which is equal to the cost of the ephemeral subgraph operation multiplied by all the potential versions of each node in the subgraph.

From Table 2, we can deduce that our approach uses the least space required w.r.t the total number of stored changes in the sequence and is asymptotically space-optimal (we do not take into account all multiplicative constant factors as well as additive factors). For the aforementioned methods, one could implement the various operations shown in Table 2 by employing additional indexing techniques (e.g., hash table with pointers to different time instances of vertices in each snapshot in the COPY method) and the stated complexities may change based on such decisions. For ephemeral operations, we assume that the respective reconstructed object is returned (e.g., the vertex as seen at time t). For versioned operations in the time interval $[t_s, t_e]$, the reconstructed ephemeral object is returned at time t_s along with a list of changes up until time t_e. Finally, to compare the performance w.r.t. updates we focus only on insertions of edges. Although different, similar performance is achieved for vertex insertions. Comparing the access times between TGI, G^* and HiNode for all operations, we can observe that HiNode is only either by multiplica-

tive or additive logarithmic factors away from the OPTIMAL access times with the exception of ephemeral 1-hop, where there is an additive A factor, that corresponds to the size of the ephemeral node. When compared to G^*, it seems that HiNode is worse by a multiplicative logarithmic factor in the Snapshot, Ephemeral and Versioned Subgraph operations without taking into account the additive factors of G^* in these operations, thus leading to various tradeoffs in efficiency. When compared to TGI, HiNode always exhibits better complexities.

5 Query Processing

In this section, we start by discussing snapshot materialization and graph traversal, which are fundamental steps in historical queries. Then, we discuss various versions of shortest paths based on the underlying notion of time. Finally, we refer to graph sampling as an argument in favor of efficiency in local queries.

5.1 Core Algorithms for Global Queries

We begin by outlining snapshot materialization of the graph at a specific time instance and how to execute a graph traversal algorithm (e.g. DFS/BFS) for a given source vertex and time instance.

Theorem 2 *Given $\mathcal{G}$ we can materialize a specific snapshot $G_t = (V_t, E_t)$ at time instance t in $O(|V_t| \log_B m + \frac{\mathcal{S}}{B})$ time (I/Os) where $\mathcal{S}$ is the total size (attributes and edges) of all vertices in V_t.*

Proof Creating a snapshot in HiNode is only a logarithmic factor away from the optimal while it compares favorably with TGI and G^* (see Section 4.4). We begin by executing a stabbing query on $\mathcal{I}$ to retrieve all the vertices that exist on the time instance t. Afterwards, we perform a `ReadVertex` operation on each of the returned diachronic nodes to obtain the final result. The initial stabbing query requires $O(\log_B m + |V_t|/B)$ time and each subsequent `ReadVertex` operation takes $O(\log_B m + |u_t|/B)$ time where $|u_t|$ is the size of the vertex u at time t. Let $\mathcal{S}$ be the total size of all vertices in V_t. Since there are $|V_t|$ `ReadVertex` operations and $\mathcal{S} \geq |V_t|$, this brings the total time to $O(|V_t| \log_B m + \frac{\mathcal{S}}{B})$. □

For DFS, the straightforward way would be to materialize the snapshot and run DFS on the snapshot. However, by using a stack data structure and repeatedly using `ReadVertex` in each newly-visited node we can naturally apply DFS on the diachronic nodes without having to resort to materialization. We only have indications as to whether this is more efficient than materializing the corresponding snapshot. In particular, preliminary results, as indicated in the next section show that in principle materialization has larger costs (e.g. requires the explicit storage of the snapshot) and it is expected that HiNode will

be more efficient as indicated in Section 6.3. Additionally, experimental results suggest that when more than one snapshots are required (a time interval and not a time instance) our method is certainly more efficient.

Theorem 3 *Given $\mathcal{G}$ we can perform a depth-first search on a specific snapshot $G_t = (V_t, E_t)$ at time instance t starting from a source vertex v in $O(|V_t| \log_B m + \frac{\mathcal{S}}{B})$ time (I/Os) where $\mathcal{S}$ is the total size (attributes and edges) of all vertices in V_t.*

Proof To perform a depth-first search an external stack data structure [28] is required. The external stack is the external memory equivalent of an internal memory LIFO data structure and supports insertions (push) and deletions (pop) in $O(1/B)$ amortized time. As a first step, we retrieve the diachronic node of v using $\mathcal{B}$ in $O(\log_B m)$ time and push the node to the stack. We then iteratively pop the node in the top of the stack, mark it as visited and perform a `ReadVertex` on the acquired node. For each outgoing edge we push the respective node in the stack and repeat the same procedure until the stack is empty. The worst case for this algorithm occurs when G_t is connected and thus all $|V_t|$ vertices are eventually inserted and deleted from the stack.

The push and pop require $O(|V_t|/B)$ amortized time in total, while all the `ReadVertex` operations require $O(|V_t| \log_B m + \frac{\mathcal{S}}{B})$ time (Theorem 2), which brings the total cost to $O(|V_t| \log_B m + \frac{\mathcal{S}}{B})$ time. □

Finally, we move to the discussion of finding the shortest path from a single source v to a single target u. We base our algorithms on the well-known Dijkstra [10] algorithm for solving this problem which stops as soon as it discovers the shortest path between these two nodes. We provide two different versions of the problem based on the underlying notion of time: a) *time travel shortest paths* and b) *time instance shortest paths*. In the former variant, the user provides the source node v and the target node u as well as a time interval $[t_s, t_e]$ and asks the shortest path between v and u that uses edges and nodes valid throughout the time interval $[t_s, t_e]$. This means that the shortest path is allowed to use edges and nodes that do not coexist at the same time instance (we allow for time traveling when traversing the shortest path - e.g. in the case of airline travel with connection cities, a traveler is usually not interested in all the flight paths coexisting at the same time). In the latter variant, the user provides again the same input but expects to find a shortest path between v and u that is valid at a particular time instance within the time interval $[t_s, t_e]$, that is time travel is not allowed. For the latter variant, one could construct all the $t_e - t_s + 1$ snapshots and run the Dijkstra algorithm in each one of them reporting at the end the one with the minimum length, which is a clear waste of time when these paths have a lot in common. In the time travel shortest path, materialization does not lead to a straightforward algorithm.

Theorem 4 *Given $\mathcal{G}$, $[t_s, t_e]$ and nodes v and u, we can find a time travel shortest path between v and u within the time interval $[t_s, t_e]$ in $O((|E_{s,e}| + |V_{s,e}| \log |V_{s,e}|) \log_B m)$ time (I/Os), where $V_{s,e}$ and $E_{s,e}$ are the set of nodes and edges respectively that are valid in the time interval $[t_s, t_e]$.*

Algorithm 5 TimeTravelSSSP($\mathcal{G}$, v, u, $[t_s, t_e]$)

Input: evolving graph sequence $\mathcal{G}$, time interval $[t_s, t_e]$, source v and target u
Output: The time travel shortest path p from v to u in the range $[t_s, t_e]$

```
    ▷ dist[w] stores the minimum distance from v to w
    ▷ prev[w] stores the predecessor of w
    ▷ Q is an external memory priority queue
 1: Using I acquire the list L of nodes that are valid in [t_s, t_e] (simple pointers to them)
 2: for each vertex w ≠ v in L do
 3:     dist[w] ← +∞
 4:     initialize the predecessor of w: prev[w] ← NULL
 5:     InsertKey(Q, w, +∞)
 6: end for
 7: while target u has not been reached do
 8:     w ← ExtractMin(Q)
 9:     for each valid neighbor z of w in [t_s, t_e] do
10:         Let Λ be the set of lengths of valid edges (w, z)
11:         Let length(w, z) = min Λ
12:         newd ← dist[w] + length(w, z)
13:         if newd < dist[z] then
14:             dist[z] ← newd
15:             prev[z] ← w
16:             DecreasePriority(Q, z, newd)
17:         end if
18:     end for
19: end while
20: return dist[u], prev[u]
```

Proof The algorithm is the same with that of Dijkstra with the exception that we consider edges and nodes that are valid in the time interval $[t_s, t_e]$. In particular, when the algorithm considers the neighbors of a node w to update their distance, it chooses nodes that are valid in $[t_s, t_e]$. In addition, the length of the edge between v and its neighbor z is the minimum length among all edges between v and z that are valid in the time interval $[t_s, t_e]$. Note that the distance (dist) and the predecessor labels (prev) can be stored separately maintaining pointers to the index and not the diachronic nodes. See Algorithm 5 for an extensive description. In the description of the algorithm, it is the definition of the Λ set that introduces the time travel in the computation of the shortest paths. Choosing the minimum among all lengths in Λ returns at the end the time travel shortest path between v and u since any other choice would lead to a shortest path with at least the same length (this is an exchange argument for the greedy method of choosing lengths). We use standard terminology for the description of the algorithm while the shortest path can be generated by backtracking from u.

The time complexity is $O\left((|E_{s,e}| + |V_{s,e}| \log |V_{s,e}|) \log_B m\right)$, where the factor $\log_B m$ is derived from accessing each field in the diachronic node, while the rest of the time complexity comes from the Dijkstra algorithm implemented with an efficient priority queue [7]. □

As previously mentioned, materialization cannot be used in this case. Employing Algorithm 5 in the TGI would require the use of the lists of different

instances of the diachronic nodes. This would mean that all accesses would be non-local and thus it is expected that the efficiency would be deteriorated. In addition, it requires more work to find all edges that are valid in $[t_s, t_e]$ since we need to traverse a list of nodes to find them. G^* is expected to be more efficient than TGI but due to the level of indirection that we need to access for finding the edges it is expected that it will be less efficient than HiNode, especially when the time interval $[t_s, t_e]$ is not small (e.g., if it contains 3 time instances).

The time instance shortest path requires more care since we need to discover a shortest path between v and u that is valid in one time instance without resorting to materialization, which is the first thing that comes to mind and gives rise to a natural algorithm.

Theorem 5 *Given $\mathcal{G}$, $[t_s, t_e]$ and nodes v and u, we can find a time instance shortest path between v and u within the time interval $[t_s, t_e]$ in $\tilde{O}((|E_{s,e}| + |V_{s,e}| \log |V_{s,e}|) \log_B m)$ time (I/Os), where $V_{s,e}$ and $E_{s,e}$ are the set of nodes and edges respectively that are valid in the time interval $[t_s, t_e]$.*

Proof The algorithm resembles that of Dijkstra with the exception that each node is accompanied by an interval in which it is valid (always within the given interval $[t_s, t_e]$). This means that in the priority queue we do not store each node only once but as many times as the different instances of the node within the time interval $[t_s, t_e]$. A description is shown in Algorithm 6. The major difference is that we handle many copies of the same node in the priority queue that correspond to many different time intervals (but not necessarily time instances). In addition, another thing to notice is that the stopping condition is not related to the target u but requires the priority queue Q to be empty. This is because when a node is processed it does not mean that it is processed for its full range of its valid interval but only for a subinterval. One could also define as a stopping condition that all instances of u corresponding to different time intervals are visited.

The time complexity is $\tilde{O}\left((|E_{s,e}| + |V_{s,e}| \log |V_{s,e}|) \log_B m\right)$, where the factor $\log_B m$ is derived from accessing each field in the diachronic node, while the rest of the time complexity comes from the Dijkstra algorithm. The $\tilde{O}$ notation represents multiplicative factors based on the implementation of the hash maps. Finally, note that each interval corresponding to an object among the $|V_{s,e}| + |E_{s,e}|$ different objects creates only $O(1)$ new records in the data structures used in Algorithm 6. □

5.2 Graph Sampling: A local query case-study

We use graph sampling [2,15] as a means to show that local operations on graphs are of critical importance. Thus, there is a need to be able to access efficiently subgraphs of a graph without resorting to materialization. Graph sampling is a technique related to picking a subset of vertices and/or edges

Algorithm 6 InstanceSSSP($\mathcal{G}$, v, u, $[t_s, t_e]$)

Input: evolving graph sequence $\mathcal{G}$, time interval $[t_s, t_e]$, source v and target u
Output: The time instance shortest path p from v to u in the range $[t_s, t_e]$

```
    ▷ dist and prev are hash maps
 1: Using I acquire the list L of nodes that are valid in [t_s, t_e]
 2: for each vertex w ≠ v in L do
 3:     dist[w_[0,+∞]] ← +∞ dist is a hash map
 4:     prev[w_[0,+∞]] ← NULL
 5:     InsertKey(Q, w_[0,+∞], +∞)
 6: end for
 7: while priority queue Q is not empty do
 8:     w_[t,t'] ← ExtractMin(Q)
 9:     for each neighbor z_[t_z,t'_z] of w such that [t,t'] ∩ [t_z,t'_z] ≠ ∅ do
10:         if t ≤ t_z ≤ t' ≤ t'_z then
11:             newd ← dist[w_[t,t']] + length(w_[t,t'], z_[t_z,t'])
12:             oldd ← dist[z_[t_z,t'_z]]
13:             if newd < oldd then
14:                 Remove z_[t_z,t'_z] from dist and prev
15:                 Add to dist, z_[t_z,t'] with value newd
16:                 Add to dist, z_[t',t'_z] with value oldd
17:                 Add to prev, z_[t_z,t'] with value w_[t,t']
18:                 Add to prev, z_[t',t'_z] with value w_[t,t']
19:                 RemoveKey(Q, z_[t_z,t'_z])
20:                 InsertKey(Q, z_[t_z,t'], newd)
21:                 InsertKey(Q, z_[t',t'_z], oldd)
22:             end if
23:         end if
24:         Similarly as in Lines 10-23, for the cases [t,t'] ⊇ [t_z,t'_z], [t,t'] ⊆ [t_z,t'_z] and
    t_z ≤ t ≤ t'_z ≤ t'
25:     end for
26: end while
27: return dist[u], prev[u]
```

from a given graph aiming at preserving and/or estimating certain desired graph properties. In this way, the new smaller graph is similar with respect to certain properties to the full one. Thus, an algorithm may be applied to the smaller graph to compute these properties for the full graph, leading to improved efficiency. The main motivating example for graph sampling is the lack of data (e.g., API rate limits in Twitter) or lack of resources (e.g., time) to access the data (e.g., the huge graph of all followers in Twitter). Although sampling can be tackled by optimization methods, these assume full access to the graph in the first place which as we said earlier is either not possible or time consuming. As a result, we focus only on simple approaches that are tailored to our framework. In particular, we assume that the evolution of the graph is fully stored in HiNode but due to time constraints we wish to access only a part of the graph at a particular instance or time interval in order to estimate certain properties. Note that [2] has studied graph sampling in a streaming framework which can only be seen orthogonally to our study.

The most important graph sampling techniques include Vertex Sampling (VS) and Traversal Based Sampling (TBS). Let $G = (V, E)$ be a simple graph.

In the VS technique, a subset $V' \subseteq V$ is chosen randomly as well as all edges between these vertices that belong to E, that is $E' = \{(u, v) : (u, v) \in E, u, v \in V'\}$. A major version of this technique is Vertex Sampling with Neighborhood (VSN), where initially a set of vertices $\tilde{V}$ is chosen and then E' is the set of all edges that are incident to $\tilde{V}$ while V' is the set of all vertices that are endpoints of the edges in E'. Finally, in TBS, a sampler starts with a set of initial vertices and then extends the sample by following edges from vertices already visited by employing various strategies (e.g., randomly, BFS, DFS).

We now discuss how graph sampling fits into our framework. In graph sampling one needs to access a limited number of vertices/edges bounded by a predefined budget $\mathcal{L}$ which is reduced each time an edge or a vertex is sampled. If the operation is applied at a single snapshot, then one can simply materialize this snapshot and then apply the sampling procedure to it. However, this is contradictory since it requires the full access of the instance which cancels all advantages of sampling. As a result, one cannot employ methods that store historical graphs that are based on materializing snapshots in order to support such operations. It is more appropriate in this case to materialize single vertices, which is the main strong point of vertex-centric storage techniques, like ours.

To clarify this point we look at the problem of computing the degree distribution when graph sampling is employed and comparing G^*, TGI and HiNode on such a scenario. VS and Random Walks (RW) have a pretty good performance in approximating the degree distribution of the underlying network (directed or undirected) [30] since they are unbiased estimators and their mean squared error is rather small. Estimating the degree distribution in a specific time instance for both methods requires sampling randomly vertices at the specific snapshot and then, in the case of RWs, visiting adjacent vertices. TGI [20] would first construct the snapshot at this time instance and move to the sampling process. On the other hand, G^* [23] and our method would only process the vertices that are sampled. This is more efficient when the budget $\mathcal{L}$ is small (e.g., $\leq 10\%$ of the size of the sampled graph). When comparing G^* and our solution, G^* is expected to be slightly more efficient in the case of single instance VS or RW since decoding each vertex for a single time instance is usually simpler (although it has the intermediate level of indirection which HiNode avoids). On the other hand, our solution is expected to be more suitable in the case of intervals of time instances since for each vertex, G^* requires more I/Os than HiNode.

In a slightly different scenario, let us assume that we wish to find the degree distribution of a graph in a given time interval. For the VS method, TGI finds each vertex by reconstructing the snapshot and then it uses the version chains that connect all instances of a vertex in a list to speed up the processing. Apart from the obvious problem with the reconstruction of the snapshot when the budget $\mathcal{L}$ is small, the vertex chains are not packed together and thus do not exhibit space locality. On the other hand, G^* uses the same idea as in the case of sampling ephemeral vertices but because of the fact that all historical information of a vertex is simply packed in a block it is easy to access

it. However, the high access cost remains due to the complicated indexing mechanism to access each time instance. Our solution has the simplest indexing mechanism while at the same time maintains all the history of the vertex within a single object and thus exhibiting high space locality leading to fast access times although more operations are required to decode the information within this diachronic node (note that decoding of a diachronic node is carried out in main memory).

5.3 Practical Considerations

The solution we proposed in Section 4 makes extensive use of the external interval tree data structure in order to provide efficient asymptotic bounds. We can take advantage of the fact that in practice, the size of an individual diachronic node and the count of the intervals it maintains for its attributes and edges is substantially small. In practice, we can replace B-trees with linked lists and hash maps since we then avoid the constant factors that arise from the use of a more elaborate data structure.

More specifically, we can use a hash map to represent $\mathcal{B}$ and omit representing $\mathcal{I}$ to reduce the space overhead (to recreate a specific snapshot we simply visit all diachronic nodes through $\mathcal{B}$). Additionally, in each diachronic node v, we omit $\mathcal{I}_v$ and replace B_v with three hash map data structures (one for each set of attributes, incoming and outgoing edges respectively). Finally, we model each of the A_v^f trees corresponding to attributes with a linked list that maintains intervals. In practice, these modifications improve our runtime efficiency at the expense of not strictly following the asymptotic guarantees of Theorem 1 regarding the time complexity. Space usage is even better w.r.t. constant factors since there is no use of elaborate data structures. Note that the above modifications are predominantly suited for the case of historical graphs that do not exhibit a specific update pattern. In extreme cases of sequences that perform a very large amount of updates on a very small subset of vertices the use of linked lists may not be suitable.

Finally, our solution could be simplified considerably if only transaction time (rather than both transaction and valid time) was considered. In particular, the diachronic node would be structured as a simple list of updates in the node. The order of the changes in the list is dictated by the order of updates. This small change in the structure of the node allows for the more efficient decoding of the diachronic node making it potentially more efficient than the decoding needed in G^*.

6 Experimental Evaluation

In this section, we provide experimental results after incorporating *HiNode* in the G^* parallel graph processing system.[3] We aim to show (i) actual space

[3] The source code is available at `https://github.com/hinodeauthors/hinode`

savings, (ii) real execution times for local queries and (iii) real execution times for global queries, for which a vertex-centric approach may be thought to be inefficient. The experiments ran on a private cluster with 21 virtual machines (VMs). 20 VMs played the role of the G^* workers, each having 1 VCPU, 5 GB RAM and 100 GB storage. One VM served as both G^* master and worker having 4 VCPUs, 28 GB RAM and 500 GB storage. All the VMs were connected through a 1GBit local network. The memory is large enough so that it can hold any diachronic node in its entirety.

6.1 Implementation Details

In the original G^* system, the indexing model is based on maintaining "(vertexID, diskLocation)" pairs for the vertices stored in each G^* server. These pairs are stored in collections that are formed in an efficient way so that the overall space required by the index is reduced and is able to fully (or partially) fit in the internal memory of each server. While the index is maintained in memory, the vertices or edges and their attributes are stored on disk. A query on the G^* system is converted to a structure of graph operators that are computed in a pipelining fashion. The basis of the graph operators is the `vertex` operator that retrieves a particular instance of a vertex along with its attributes and edges from the disk. Note that allowing for the index to reside in memory does not provide HiNode or G^* with some advantage since the comparison is only between these two storage models.

We incorporate our work into the G^* system by replacing the existing indexing model with the model proposed in Section 4 along with the practical improvements described in Section 5.3. The system is further modified so that it stores entire diachronic nodes on disk instead of vertices. To answer queries, the system still makes use of the `vertex` operator. However, the modified system retrieves an entire diachronic node from the disk and thus, it performs an operation equivalent to `ReadVertex` to obtain a particular instance of a vertex. Note that in our prototype implementation we don't examine partitioning issues and all the experiments are run under the same partitioning policies for both G^* and HiNode.

6.2 Dataset Description

We use both real and synthetic datasets. The former provide insights into actual performance benefits, while the latter allow us to evaluate our approach under a wide range of configurations.

The real datasets were obtained from the Large Network Dataset Collection of SNAP [24]. The first dataset is a citation graph of the arXiv hep-th category released as a part of the 2003 KDD Cup [13]. The dataset contains citations from January 1993 to April 2003, which we use to create a sequence of monthly snapshots of the citation graph. The second dataset is similar to the first

Table 3 Experiments on real datasets. The number of vertices and edges refer to the last snapshot of the sequence

Dataset	Vertices	Edges	Snapshots
hep-Th	27770	352807	156
hep-Ph	34546	421578	132
US Patents	3774768	16518948	444

dataset as it focuses on the arXiv hep-ph category on the same time period while featuring a slightly larger count of vertices and edges. In both datasets, we omit 0.4% of the total edges due to the difficulty of mapping them to a specific snapshot (e.g. paper A cites paper B but paper B is inserted in the dataset with a later timestamp than that of A). The last dataset maintains records for all the US utility patents granted between 1963 and 1999 and their cross-citations. We build a sequence of monthly snapshots for that time period while omitting 0.04% of the edges due to insufficient date information in the dataset (e.g. withdrawn patents). In the real datasets we only focus on the edges between vertices and do not maintain any attributes (such as names or weights). A detailed overview of each real dataset is shown on Table 3.

The synthetic datasets follow either the Erdős-Rényi (ER) [11] model or the Barabási-Albert (BA) [4] scale-free graph model. The latter resembles real-world settings and environments more closely. To construct an ER synthetic dataset, we supply the number of vertices and edges in the first snapshot, the number of all snapshots, and the percentage of vertices and edges inserted or updated between snapshots (e.g., in a snapshot of 1000 vertices and 1000 edges an insertion rate of 5% would result in the next snapshot having 1050 vertices and 1050 edges). Vertices have a name and edges are weighted. An update is defined as the alternation of either a vertex's name or an edge's weight. BA sequences are created similarly. In a BA sequence, each newly inserted vertex is connected to the existing vertices, preferring those with a larger degree, with the number of edges created for each newly inserted vertex specified by a parameter.

Finally, we note that our datasets do not contain events related to the deletion of vertices or edges since a deletion does not essentially differ from an update. More specifically, consider an edge that is alive and that at some time instance it becomes deleted. Upon the deletion of the edge we "update" a relevant field used to mark the edge's lifetime. Any queries regarding that edge must also take into account the value stored by that field.

6.3 Snapshot Retrieval

In this section, we focus on the time required for snapshot retrieval in G^* and HiNode. More specifically, in both systems each worker is assigned an equal portion of each snapshot in the sequence through a round-robin vertex assignment. Table 4 shows the time required to retrieve either 1 snapshot, 20% of a sequence's snapshots or 40% of a sequence's snapshots for a single

Table 4 Snapshot(s) portion retrieval time (in seconds) for a single worker. The percentages correspond to the amount of retrieved snapshots in the sequence.

	1 Snapshot	20% Snapshots	40% Snapshots
G^*	7.4	20	27.5
HiNode	6.7	7.6	8.7

worker.[4] We focused on a single worker in order to cancel out the additional communication cost for reconstructing the whole snapshot, favoring in this way G^* since HiNode is more space efficient.

While both solutions perform comparably in the case of a single snapshot, HiNode performs considerably better when retrieving a range of snapshots. The former case is attributed to the fact that G^* is characterized by the intermediate level of indirection that we need to access for finding the edges. On the other hand, the latter case is due to HiNode requiring less I/Os than G^* overall since in HiNode a vertex's history is contained within its diachronic node, while in G^* we require multiple I/Os to fetch a vertex's history due to its indexing mechanism.

6.4 Local Query Evaluation

To demonstrate the efficiency of our approach in local query evaluation, we conducted experiments that focused on the main primitive operations (or a similar variant) of VS and RW (see Section 5.2) and compared our approach against G^*. In particular, we studied the time required for two-hop neighborhood retrieval of a specific vertex, as well as, vertex sampling in general.

In the first experiment, given a source vertex and a (range of) snapshot(s) the query outputs all the two-hop neighborhoods for each instance of the source vertex. We performed two-hop neighborhood retrieval for varying degree sizes and snapshot ranges and present our results in Figure 4.[5][6] HiNode improved on G^* by a factor of up to 4.2 times.

In the latter experiment, we focused on retrieving a randomly chosen set of vertices from each worker. When the query was executed on a range of snapshots, all possible corresponding instances of each sampled vertex were returned. Additionally, we executed the query for different sample sizes. In a slight modification of the experiment, instead of reporting the sample itself, we aggregated the vertices reported by each worker so as to find the degree distribution of a graph in a time instance or a time interval. This modification translates to reduced communication costs since each worker only reports the cardinality of each degree count. The time required for this query can be seen

[4] Dataset - Undirected Barabási-Albert graph: Starting vertices = $1M$, edges per newly inserted vertex = 5, vertex insertions per snapshot = $2K$, snapshots = 100

[5] Dataset - Undirected Barabási-Albert graph: Starting vertices = $1M$, edges per newly inserted vertex = 5, vertex insertions per snapshot = $20K$, snapshots = 100

[6] In the case of querying the 40% of the sequence for the two-hop neighborhood of the vertex with the largest degree, G^* was unable to finish since it run out of memory.

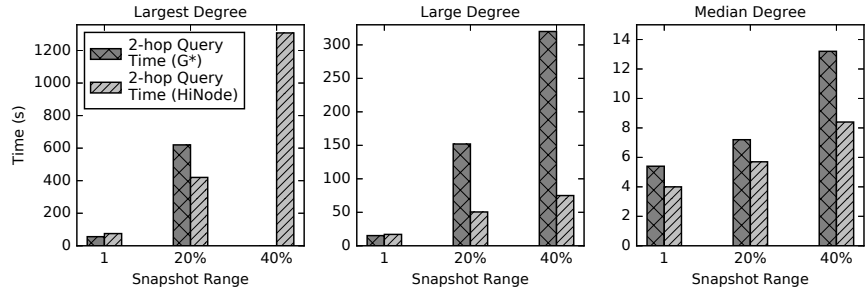

Fig. 4 Two-hop neighborhood query time.

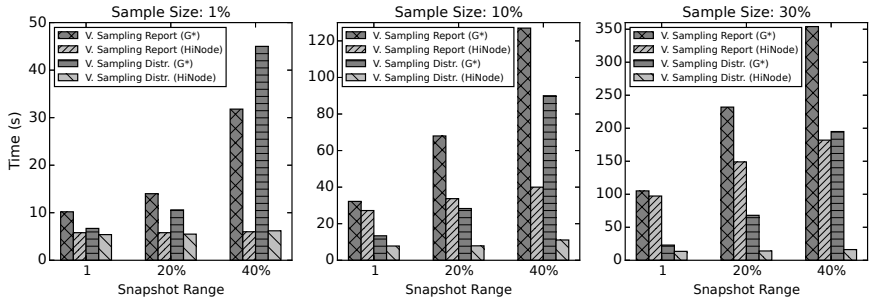

Fig. 5 Vertex sampling query time. The dataset used is the same as in Figure 4.

in Figure 5. In this experiment, HiNode improved upon G^* by a factor of up to 12 times, i.e., the improvement is by an order of magnitude.

Both experiments show that in the case of a single snapshot the two systems have comparable performance, whereas our approach is substantially better than G^* when the query is concerned with a range of snapshots. In addition, HiNode performs increasingly better on nodes with a higher degree. These two observations stem from the fact that HiNode requires less I/Os than G^*, to retrieve all the instances of a vertex.

6.5 Space Consumption

Table 5 shows the space utilization of each system for each of the real datasets. The space savings are up to an order of magnitude. Our proposed solutions use approximately 90% less space for indexing and 76% to 88% less space for the data files. Recall that in our solution the index consists of a `LinkedHashMap` containing "(DiacNodeID, Location)" pairs.

Next we experiment with the synthetic datasets for several sequence sizes and insertion/update rates, as shown in Figure 6. Our main observations are as follows. (i) As previously, the space savings reach an order of magnitude. (ii) The higher the insertions or updates, the more significant the savings. This can

Table 5 Space consumption in real datasets

	G* Storage Module Size (MB)		HiNode Storage Module Size (MB)		Difference (%)	
Dataset	Index Size	Data Size	Index Size	Data Size	Index Size	Data Size
hep-Th	9.49	788.06	0.95	98.63	-89.99%	-87.48%
hep-Ph	12.33	859.5	1.17	102.81	-90.51%	-88.04%
US Patents	1094.41	23407.75	122.38	5456.63	-88.82%	-76.69%

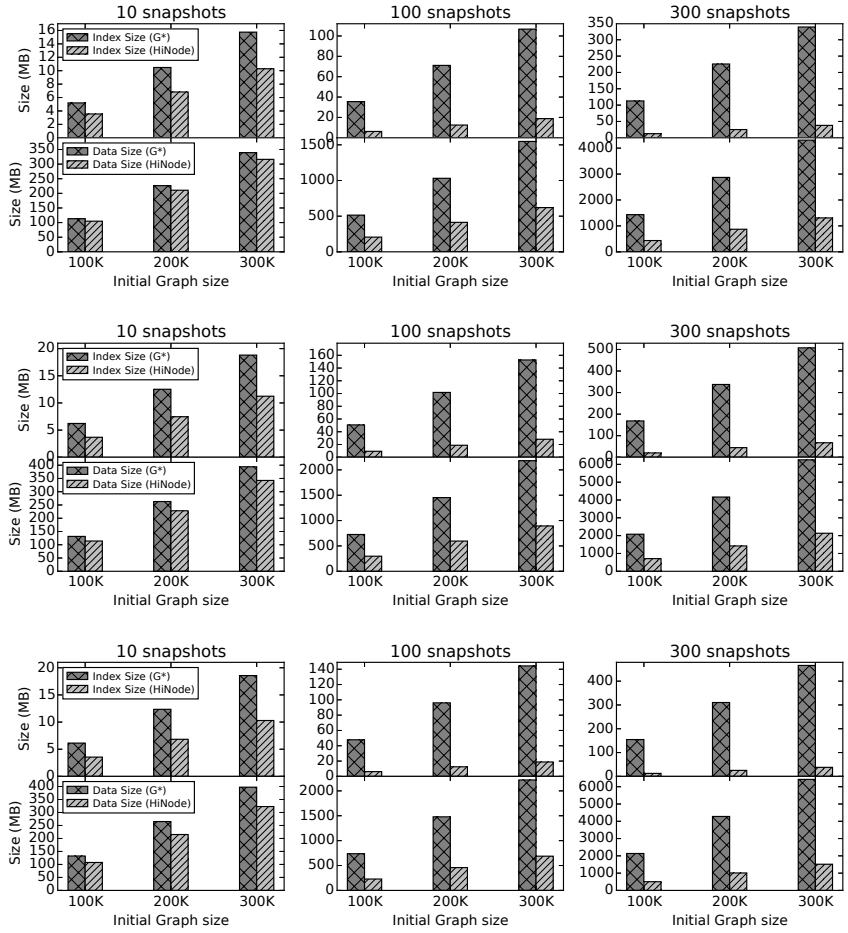

Fig. 6 Space consumption in sequences with $insertion\_rate = update\_rate = 1\%$ (top), $insertion\_rate = 2\%, update\_rate = 1\%$ (middle), and $insertion\_rate = 1\%, update\_rate = 2\%$ (bottom). In each subfigure, the upper part refers to the index size, and the lower to the size of the data files.

be explained by the fact that, since all vertex and edge updates are stored in the diachronic nodes, the size of the index becomes larger only when new vertices are created in the sequence. A similar observation can be made about the data

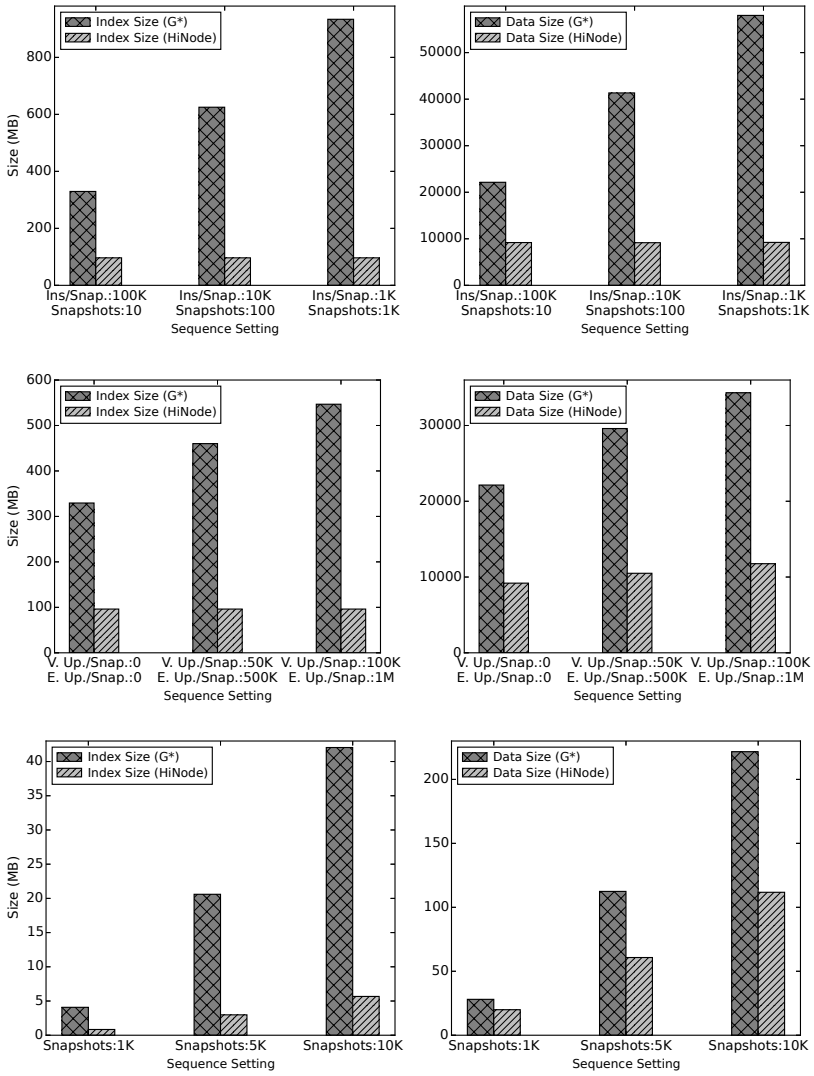

Fig. 7 Space consumption in the BA datasets. Top: effect of granularity (starting vertices $= 2M$ and edges per newly inserted vertex $= 10$). Middle: effect of updates (starting vertices $= 2M$, edges per newly inserted vertex $= 10$, insertions per snapshot $= 100K$, snapshots $= 10$). Bottom: effect of the number of snapshots (starting vertices $= 10K$, edges per newly inserted vertex $= 1$, insertions per snapshot $= 10$).

file sizes. (iii) In general, our proposed system favors sequences with a higher update-to-insertion ratio and (iv) the relative differences in space consumption remain the same across experiments with different starting vertices and edges counts. Similar observations can be drawn for BA (see Figure 7), where the savings in space are slightly smaller, i.e., up to 84% less space.

6.6 Time Efficiency for Global Queries

In the last part of the experiments, we measure the running time for the following global queries [23]:

Vertex Degree Distribution (DegDistr): for each graph snapshot, count the vertices with a specific vertex degree, sorted by the vertex degree in a descending order.

Average Vertex Degree (AvgDeg): for each graph snapshot, compute its average vertex degree, and give results in a descending order.

Clustering Coefficient Distribution (ClCoeff): for each graph snapshot, compute the clustering coefficient of its vertices. Report the number of vertices grouped by the clustering coefficient sorted in a descending order. Note that this is a bulk synchronous parallel (BSP) operator and more difficult to evaluate.

Single Source Shortest Path Distance Distribution (ShortPath): for each graph snapshot, compute the distance from a source vertex to all other vertices in the snapshot. Report the count of vertices with a specific distance to the source vertex sorted in a descending order.

The results for the real datasets are reported in Table 6. The "DegDistr" and "AvgDeg" queries were run on all the monthly snapshots of each sequence on both systems. However, due to high memory demand by the original G^* system, running the "ClCoeff" query in all snapshots of the "US Patents" dataset was infeasible. For that reason, we applied the "ClCoeff" query in subsets of the snapshots in the "US Patents" sequence. More specifically, in Table 6 the symbols "$*$", "†" and "‡" represent that the query was run on the last snapshot of the sequence, the last five snapshots of the sequence and all the snapshots, respectively. The first observation that can be made is that our system is more efficient for the "DegDistr" and "AvgDeg" queries, yielding up to 30% faster response times. This can be explained by the fact that, since we retrieve entire diachronic nodes from the disk, we need to make fewer accesses on the secondary memory compared to the original system which retrieves specific instances of each vertex. These benefits outweigh the additional time overhead of reconstructing a vertex from a diachronic node in a particular time instance, thus reducing the total time cost. In the "ClCoeff" query our system has slightly inferior performance compared to the original system that can be explained by the nature of the datasets. The datasets exhibit a "cold start" phenomenon in that the first snapshots of the sequence have very few vertices and edges that in turn results to the cost of vertex reconstructions overcoming the gains of the fewer disk accesses. This is also shown in the "US Patents" dataset, where our system has better performance when the "ClCoeff" query focuses on the (quite large) five last snapshots of the sequence.

We achieve significant speedups for the synthetic datasets as well. The results for the ER sequences are shown in Figure 8. For all three types of queries, the maximum reduction in response time is 54%-56%.

Table 6 Time efficiency in real datasets

	G* System Time (s)			HiNode System Time (s)			Difference (%)		
Dataset	DegDistr	AvgDeg	ClCoeff	DegDistr	AvgDeg	ClCoeff	DegDistr	AvgDeg	ClCoeff
hep-Th	7.9	7.2	723	7.7	6.2	855	-2.50%	-13.88%	18.26%
hep-Ph	9.7	8.4	410	7.7	7.2	471	-20.61%	-14.28%	14.87%
US Patents	329	316	* 213 † 496 ‡ -	242	221	* 234 † 377 ‡ 1204	-26.44%	-30.06%	* 9.85% † -23.99% ‡ -

Next, we tested the BA synthetic sequences, executing the queries on varying sequence portions. More specifically, we executed a query on the last snapshot of the sequence or on a selection of the last 5-20% of the snapshots of the sequence. Additionally, we ran the queries on non-consecutive snapshots by specifying an appropriate step size. Initially, we investigated the impact of granularity, as in the space-efficiency experiments. We ran the three queries in sequences of 10, 100 and 500 snapshots. Regarding the sequence with the 10 snapshots, since the percentages of the previous paragraph do not directly correspond to meaningful snapshot ranges, we ran the queries in the last 1, 2 and 5 snapshots. The results can be seen in Figure 9. In the case of querying only the last snapshot of the sequence, our method is slower since it suffers from the time overhead of reconstructing that particular snapshot. However, the time efficiency of our approach improves as the query percentage becomes higher. The effect of updates is shown in Figure 10. Again, for higher query percentage we achieve better performance.

Finally, we evaluated the impact of graph density on the total query time, after building sequences of varying vertex degree per newly inserted vertex. The results showed that density played no significant role as the difference remained relatively the same. More specifically, Figure 11 shows the evaluation of "ShortPath" for a sequence of 100 snapshots with varying vertex degree. Since the running time of "ShortPath" is highly dependant on the network structure and the selection of the source vertex, we ran "ShortPath" on BA graphs (connected graphs) with the vertex with the largest degree acting as the source vertex. It can be seen that while G^* is better in the case of querying a single snapshot, the two systems are comparable in the case of querying 5% of the snapshots of the sequence and HiNode outperforms G^* when querying the 10% of the snapshots of the sequence.

7 Discussion

Multiple Universes HiNode is designed so that it can support transaction time as well as valid time. Additionally, with minor modifications, HiNode can also support multiple universes in the sense that the history has a tree-like shape. This is reminiscent of the notion of full persistence in data structures [31], in the sense that the history of the data structure is fully characterized by a tree structure. Similarly, since history is not linear, we require its explicit representation by a *history tree*. Instead of talking about time that implies a linear evolution we now talk about versions of graphs. New version instances

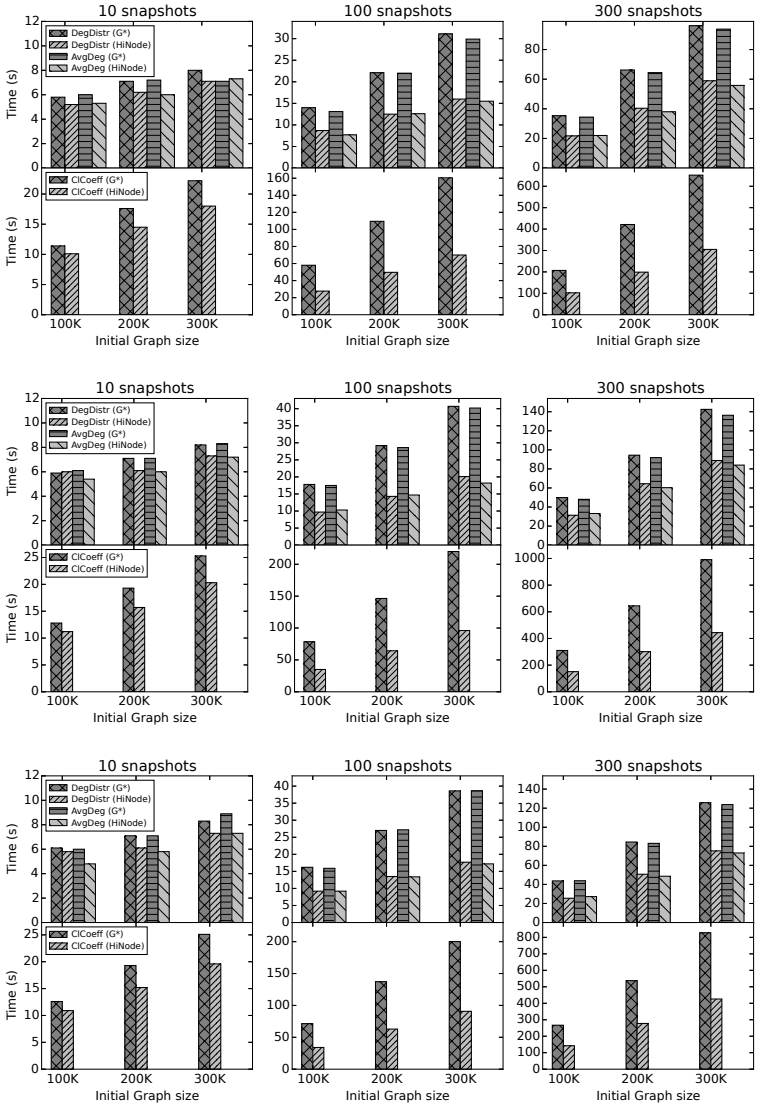

Fig. 8 Time efficiency in sequences with $insertion\_rate = update\_rate = 1\%$ (top), $insertion\_rate = 2\%, update\_rate = 1\%$ (middle), and $insertion\_rate = 1\%, update\_rate = 2\%$ (bottom).

are created by making updates to existent versions of the history tree. For example, let a node v of the history tree corresponding to the graph of version v. An update at version v gives a new instance that is represented by node u (with version u) that is a child of v in the version tree. We refer to the interested reader for a more detailed analysis to [31]. The crucial point is that navigation in history requires the efficient support of nearest common ancestor

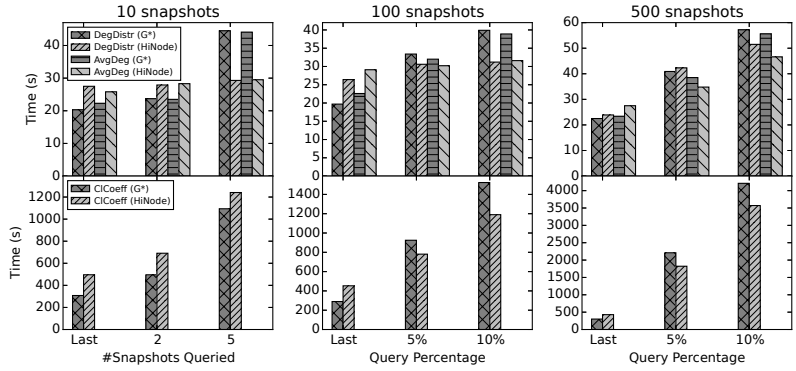

Fig. 9 Effect of granularity on time. Starting vertices = $1M$, edges per newly inserted vertex = 10. Insertions/snapshot = $200K/20K/4K$ for sequences with 10/100/500 snapshots, respectively.

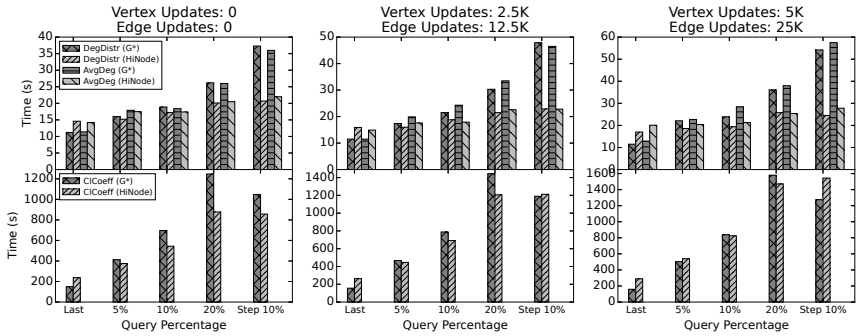

Fig. 10 Effect of updates on time. Starting vertices = $1M$, edges per newly inserted vertex = 5, snapshots = 10.

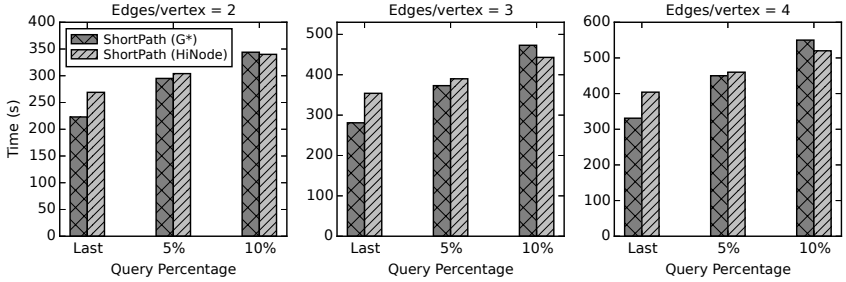

Fig. 11 Shortest path evaluation and effect of graph density. Starting vertices = $1M$, vertex insertions per snapshot = $20K$, snapshots = 100. The source vertex was the vertex with the highest degree.

queries on the history tree. In this way, searching for a version v in a node is

equivalent to finding the version u that exists in this node and is the nearest ancestor among all versions in the node of v.

HiNode can support such a tree like history by employing an external memory data structure that can answer efficiently such nearest ancestor queries (see [8]). Having such a structure requires no other changes to our structure. G^* cannot support efficiently a tree-like history even in the presence of a history tree since the structure of the diachronic nodes and the index need to change considerably. TGI cannot support such a notion of time since its snapshot architecture is rather incompatible with it.

Registering Updates An issue of our approach is the silent assumption that updates between two instances (snapshots) are readily available. Although this is trivial for a data owner (e.g. Facebook), this is not the case for users that have access to two successive instances but not to the real updates between them. Thus, the problem is that given two instances of the graph at time t and $t+1$ we must discover the differences in the nodes and the edges between these two instances in order to register them in the respective diachronic nodes.[7]

We assume for simplicity that each node has a unique identifier that remains invariant in history. In this case, HiNode picks each node at time instance $t+1$ and finds it in instance t comparing them and registering the differences in the respective diachronic nodes. For all nodes in $t+1$ that are not found in t we create new diachronic nodes. The time complexity for this procedure between two time instances is $O(m)$ provided that we can search for the identifier of a node in $O(1)$ time by using hashing, where m is the total number of changes stored in HiNode. Finally, in the extreme case where the identifiers are not invariant over the course of history and even in the case where they are reused we can always apply the same approach with some additional space consumption. In particular, if an existent node at time t with identifier v has its identifier changed at time $t+1$ to v' then a new diachronic node is created that corresponds to this node. Although we use more space to save this node, since we have no other means of identifying this phenomena rather than resorting to graph isomorphism techniques, the results of any queries of the user will always be correct. Even in the case, where a node that at time $t+1$ has the identifier of another node at time t will be stored correctly since all its info will be registered in the diachronic node.

8 Conclusions

We advocate employing a vertex-centric approach to storing the evolving history of graphs. We show that this leads to an asymptotically space-optimal solution, which is efficient for both local and global queries. Local queries benefit from the fact that only the vertices of interest are retrieved instead of entire snapshots. Global queries can also benefit from the fact that fewer

[7] We would like to thank an anonymous reviewer for pointing out this issue.

disk accesses are required, despite the overhead of snapshot reconstruction, as shown in real runs in the G^* parallel graph processing system. In our experiments, the space and time savings were up to an order of magnitude. Finally, apart from a qualitative comparison of our approach against existing solutions, we discuss how we support non-linear tree-like time modelling (in addition to valid and transaction time notions) and we how we efficiently derive snapshot differences.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. N. K. Ahmed, J. Neville, and R. Kompella. Network sampling: From static to streaming graphs. *ACM Transactions on Knowledge Discovery from Data*, 8(2):7, 2014.
3. L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.
4. A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
5. Cassovary. "big graph" processing library. `https://github.com/twitter/cassovary`.
6. N. R. Brisaboa, D. Caro, A. Fariña, and M. A. Rodríguez. A compressed suffix-array strategy for temporal-graph indexing. In *SPIRE*, pages 77–88, 2014.
7. G. S. Brodal and J. Katajainen. Worst-case external-memory priority queues. In *SWAT*, pages 107–118, 1998.
8. G. S. Brodal, K. Tsakalidis, S. Sioutas, and K. Tsichlas. Fully persistent B-trees. In *SODA*, pages 602–614, 2012.
9. D. Caro, M. A. Rodríguez, and N. R. Brisaboa. Data structures for temporal graphs based on compact sequence representations. *Information Systems*, 51:1–26, 2015.
10. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
11. P. Erdős and A. Rényi. On random graphs. I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.
12. J. Gao, C. Zhou, and J. X. Yu. Toward continuous pattern detection over evolving large graph with snapshot isolation. *The VLDB Journal*, 25(2):269–290, 2016.
13. J. Gehrke, P. Ginsparg, and J. M. Kleinberg. Overview of the 2003 KDD cup. *SIGKDD Explorations*, 5(2):149–151, 2003.
14. A. Giraph. `http://giraph.apache.org/`.
15. P. Hu and W. C. Lau. A survey and taxonomy of graph sampling. *CoRR*, abs/1308.5865, 2013.
16. W. Huo and V. J. Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *SSDBM*, pages 38:1–38:4, 2014.
17. U. Kang, H. Tong, J. Sun, C. Lin, and C. Faloutsos. GBASE: a scalable and general graph management system. In *SIGKDD*, pages 1091–1099, 2011.
18. U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: mining peta-scale graphs. *Knowledge and Information Systems*, 27(2):303–325, 2011.
19. U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, pages 997–1008, 2013.
20. U. Khurana and A. Deshpande. Storing and analyzing historical graph data at scale. In *EDBT*, pages 77–88, 2016.
21. G. Koloniari, D. Souravlias, and E. Pitoura. On graph deltas for historical queries. *WOSS*, 2012.
22. A. Kosmatopoulos, K. Giannakopoulou, A. N. Papadopoulos, and K. Tsichlas. An overview of methods for handling evolving graph sequences. In *ALGOCLOUD*, pages 181–192, 2015.
23. A. G. Labouseur, J. Birnbaum, P. W. Olsen, S. R. Spillane, J. Vijayan, J. Hwang, and W. Han. The G* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases*, 33(4):479–514, 2015.

24. J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.
25. G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.
26. E. M. McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles, 1980.
27. J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. In *SIGMOD*, pages 145–156, 2012.
28. R. Pagh. Basic external memory data structures. In *Algorithms for Memory Hierarchies*, pages 14–35, 2002.
29. C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. *PVLDB*, 4(11):726–737, 2011.
30. B. F. Ribeiro and D. Towsley. On the estimation accuracy of degree distributions from graph sampling. In *CDC*, pages 5240–5247, 2012.
31. B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.
32. K. Semertzidis, E. Pitoura, and K. Lillis. Timereach: Historical reachability queries on evolving graphs. In *EDBT*, pages 121–132, 2015.
33. B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD*, pages 505–516, 2013.
34. S. R. Spillane, J. Birnbaum, D. Bokser, D. Kemp, A. G. Labouseur, P. W. Olsen, J. Vijayan, J. Hwang, and J. Yoon. A demonstration of the G* graph database system. In *ICDE*, pages 1356–1359, 2013.
35. Y. Yang, J. X. Yu, H. Gao, J. Pei, and J. Li. Mining most frequently changing component in evolving graphs. *World Wide Web*, 17(3):351–376, 2014.

A The WriteAttribute Cases

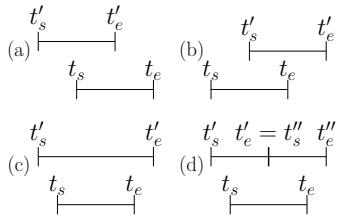

Fig. 12 Cases of existing intervals for the field f

We analyze the two possible cases in `WriteAttribute`. In the first case, the `field` f does not have any values associated with it in the time interval $[t_s, t_e]$. In that case we proceed as follows: We insert a quadruple $(f, \{\ell_1, \ell_2, \ldots\}, t_s, t_e)$ in $\mathcal{I}_v$. In addition, a record $(\{\ell_1, \ell_2, \ldots\}, t_s, t_e)$ is stored in f's respective B-tree A_v^f.

In the second case, the `field` f has values associated with it in the time interval $[t_s, t_e]$, i.e. there exist (up to) two intervals $[t'_s, t'_e]$ and $[t''_s, t''_e]$ in the data structure, such that either (a) $t'_s < t_s < t'_e < t_e$, (b) $t_s < t'_s < t_e < t'_e$, (c) $t'_s < t_s < t_e < t'_e$ or (d) $t'_s < t_s < (t'_e = t''_s) < t_e < t''_e$ is true (Figure 12). In that case, we search $\mathcal{I}_v$ for $[t'_s, t'_e]$ corresponding to the field f (and $[t''_s, t''_e]$ if it exists) by simulating an insertion of this interval in $\mathcal{I}_v$. Let $v_{t'}$ be the node of $\mathcal{I}_v$ that interval $[t'_s, t'_e]$ is to be stored. After locating the at most three lists in which it is to be stored we search these lists based on the endpoints of $[t'_s, t'_e]$. If there are more than one such intervals then we use the identifier of $[t'_s, t'_e]$ to search among them and locate this interval. The same procedure is applied for $[t''_s, t''_e]$.

Afterwards, we perform a series of interval insertions and deletions in $\mathcal{I}_v$ and the corresponding A_v^f B-tree depending on the subcases presented below (the resulting intervals end up with the appropriate set of values based on their original intervals):

Subcase (a) Deletion of $[t'_s, t'_e]$ followed by the insertion of $[t'_s, t_s)$, $[t_s, t'_e)$ and $[t'_e, t_e]$
Subcase (b) Deletion of $[t'_s, t'_e]$ followed by the insertion of $[t_s, t'_s)$, $[t'_s, t_e)$ and $[t_e, t'_e]$
Subcase (c) Deletion of $[t'_s, t'_e]$ followed by the insertion of $[t'_s, t_s)$, $[t_s, t_e)$ and $[t_e, t'_e]$
Subcase (d) Deletion of $[t'_s, t'_e]$ and $[t''_s, t''_e]$ followed by the insertion of $[t'_s, t_s)$, $[t_s, t'_e)$, $[t''_s, t_e)$ and $[t_e, t''_e]$