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

Andreas Kosmatopoulos, Kostas Tsichlas, Anastasios Gounaris
Aristotle University of Thessaloniki, Greece

Spyros Sioutas
Ionian University
Corfu, Greece

Evaggelia Pitoura
University of Ioannina
Ioannina, Greece

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 approach and we propose a storage model that is more space efficient than any other proposal to date, and, moreover, is asymptotically space-optimal. We also describe its basic operations and applications. Furthermore, 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 improvements up to an order of magnitude.

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 originally representing is constantly evolving with nodes and edges being inserted, updated, or deleted altogether. For example, in a social network, friendships are very frequently created and ended, while in a citation network each new work cites a selection of the already existing works.

An important challenge that arises with the presence of dynamicity in such networks (and their respective graphs) is the appropriate handling of their history so that we can extract features 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 *"what was the diameter in a group of friends between 2010 and 2012"* in social networks or *"how many times has author A collaborated with author B in 2000"* in citation networks.

A key aspect in effectively tackling the overall problem is the proper organization of the graph's states at different time instances, i.e. the organization of the graph's *snapshots*. Additionally, the sets of all actions that occur between two snapshots (node/edge insertions, deletions and updates), which are commonly referred to as *deltas*, should be maintained in a practical fashion, too. A simple approach to solving the problem would be to explicitly store all snapshots separately. However, a system could achieve significant performance improvements by maintaining an indexing and storage mechanism that takes advantage of the fact that there are nodes 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 of *deltas* should also be designed with their applications in mind. The application queries can be distinguished between (i) *local* queries, which require information of a single graph vertex or a few vertices, and (ii) *global* queries, which require information about most or all vertices. 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 node A at time point 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 is that it moves away from the concept of using deltas to reconstruct specific snapshots, e.g., as in [23, 16, 14, 15], and can be outlined as follows. Each unique graph node has a history in the graph sequence which is defined by the sequence of snapshots in which it exists. This history is stored within the node in a space-optimal manner, and is structured in such a way so that various queries can be supported (e.g., reconstruction of a specific snapshot). This local storage of history allows for straightforward local handling of nodes at various time instances or intervals. To accomplish this, we look at all this history within the node as time inter-

vals where various geometric operations, such as stabbing queries, are to be supported. These geometric operations in fact implement a fundamental set of access operations on the snapshots of the sequence. Additionally, we experimentally show that our technique is also particularly efficient for global queries as well, especially when a range of snapshots is considered, because the overheads of 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 each node can be stored in its entirety in a machine's memory.

In summary, we make the following contributions:

- We propose the first purely *entity-centric*, and more specifically, *vertex-centric* approach to organize historic graphs. As explained in [15], a system may be organized in *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 nodes and edges and their individual history). The former is inherently more suitable for global queries, while the latter is advantageous for local queries, because it needs to reconstruct only the sub-graph of interest instead of the complete graph.

- We show that our storing and indexing approach is more space efficient than other existing approaches, and moreover, we provide a theorem that states its asymptotic space-optimality.

- We show in detail the main processing primitive operators supported, on top of which any complex historic graph execution plan can be built, e.g., traversing a specific reconstructed snapshot, finding the shortest path between two vertices or computing the clustering coefficient of the graph vertices in the a graph sequence.

- To prove the practicality of our approach, we have implemented and incorporated it in the G^* parallel graph processing system [18, 27]. 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 performance benefits can reach an order of magnitude.

The rest of this work is organized as follows. In the next section we discuss related work. In Sec. 3, we present the main definitions and notation. We introduce our storage model in Sec. 4. We present example applications and practical improvements in Sec. 5. In Sec. 6, we conduct experimental evaluation of our method and we conclude in Sec. 7.

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 [26], Pregel [20], Giraph [9] and others (e.g. GBase [12], Pegasus [13], Cassovary [4]) 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 [17].

Perhaps, the closest proposals to ours are those in [14, 18, 27]. 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 space-optimality.

In [14] 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). Their solution is 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 [15]. Similarly to the techniques in [14, 15], 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 [14, 15], and actually, is space optimal.

A parallel graph processing system named G^* has been proposed in [18, 27]. 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 nodes as well as the historical information within the nodes 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 the evaluation results, we show that we can improve on [18, 27] up to an order of magnitude.

An earlier attempt to efficiently handle historic graph queries has appeared in [23], which is later generalized in [16]. Evolving graph sequences can also be engineered to permit efficient evaluation of specific features or queries, such as historical reachability queries [25], mining the most frequently changing component [28] or shortest path distance queries [11]. Finally, in [5, 6], 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 fields on their vertices and edges (e.g. weights). In general, combining our techniques with compression ones is an interesting direction for future work.

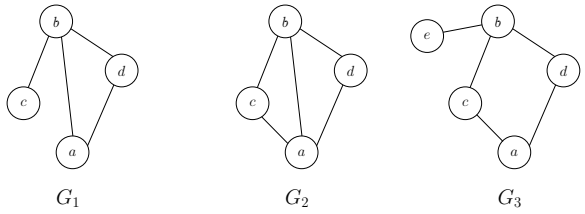

Figure 1: Evolving Graph Sequence. G_2 is obtained by inserting an edge to G_1, while G_3 is obtained by creating a node, inserting an edge and deleting an edge from G_2 respectively

3. DEFINITIONS, NOTATION AND QUALITATIVE COMPARISON

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, with each vertex or edge possessing its own set of attributes. We follow a linear notion of time and define it to be a strictly increasing quantity measured in indivisible time intervals. 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. Lastly, the notation used throughout the remainder of this work is summarised in Table 1.

In the next paragraph, 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 Δ framework introduced in [15], which is summarized as follows. An *ephemeral node* is a node at a specific time instance. Thus, one needs to specify the time instance t for which we are interested to retrieve the ephemeral node. The ephemeral node 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 node or to an edge. An ephemeral edge is similarly defined. A Δ is a set of ephemeral nodes 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 version of the graph. As a Δ, an event is simply the set of ephemeral nodes 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 nodes and edges at time instance t.

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 fields and edges of v across all of $\mathcal{G}$
$\|v_t\|$	Size of vertex v at time t
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 or field (e.g. name, weight etc.)
$\mathcal{I}_v$	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

3.1 Qualitative Comparison

Following the above definitions, we provide a qualitative comparison between different methods in Table 2, which is an extended version of the corresponding table in [15]. *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 nodes of a particular node. The OPTIMAL row corresponds to the ideal space and time access costs for the operations in the worst-case.[1]

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. One could combine these methods (COPY+LOG)

[1]For G^*, TGI and our solution, 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.

	Index Size	Edge Insertion	Snapshot	Ephemeral Vertex	Versioned Vertex	Ephemeral 1-Hop	Versioned 1-Hop	Ephemeral Subgraph	Versioned Subgraph
OPTIMAL	$\|G\|$	1	$\|S\|$	$\|A\|$	$\|C\|$	d	$d\|C\|$	$\|W\|$	$min\{\|W\|\|C\|, \|G\|\}$
LOG	$\|G\|$	1	$\|G\|$	$\|G\|$	$\|G\|$	$\|G\|$	$\|G\|$	$\|G\|$	$\|G\|$
COPY	$\|G\|^2$	$\|G\|$	$\|S\|$	$\|S\|$	$\|S\|\|C\|$	$\|S\|$	$\|S\|\|C\|$	$\|S\|$	$\|S\|\|C\|$
COPY+LOG	$\frac{\|G\|^2}{\|E\|}$	$\frac{\|G\|}{\|E\|}$	$\|S\| + \|E\|$	$\|S\| + \|E\|$	$\|G\|$	$\|S\| + \|E\|$	$\|G\|$	$\|S\| + \|E\|$	$\|G\|$
TGI	$h\|G\|$	$h\|G\|$	$h\|S\| + \|E\|$	$h\|S\| + \|E\|$	$\|C\|\|S\|$	$h\|S\| + \|E\|$	$\|C\|\|S\|$	$h\|S\| + \|E\|$	$h\|S\| + \|C\|\|W\|$
G^*	$\|G\| + \frac{\|G\|^2}{\|E\|\|C\|}$	$g + \|E\| + \frac{\|G\|}{\|E\|}$	$\|C\|\frac{\|G\|}{\|E\|} + \|E\| + g + \|S\|$	$\|C\|\frac{\|G\|}{\|E\|} + \|E\| + g + \|A\|$	$\|C\|\left(\frac{\|G\|}{\|E\|} + g\right) + \|E\|$	$\|C\|\frac{\|G\|}{\|E\|} + \|E\| + g + \|A\| + d$	$\|C\|\left(\frac{\|G\|}{\|E\|} + g\right) + \|E\| + d\|C\|$	$\|C\|\frac{\|G\|}{\|E\|} + \|E\| + g + \|W\|$	$\|C\|\left(\frac{\|G\|}{\|E\|} + g\right) + \|E\| + \|C\|\|W\|$
Ours	$\|G\|$	$g + \lg\|C\|$	$\|S\|\lg\|C\|$	$g + \|A\|\lg\|C\|$	$g + \|C\|\log\|C\|$	$g + d\lg\|C\|$	$g + d\|C\|\lg\|C\|$	$\|W\|\lg\|C\|$	$\|C\|\|W\|\lg\|C\|$
Impl.	$\|G\|$	$g + \|C\|$	$\|S\|\|C\|$	$g + \|A\|\|C\|$	$g + \|A\|\|C\|$	$g + d\|C\|$	$g + d\|C\|$	$\|W\|\|C\|$	$\|W\|\|C\|$

Table 2: 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 node in the operation Ephemeral Vertex) or to a mean value (e.g., mean size of a node in operation Versioned Subgraph). $|G| \rightarrow$ total number of stored changes; $|S| \rightarrow$ size of snapshot; $|E| \rightarrow$ eventlist size; $h \rightarrow$ height of a tree of snapshots used in TGI. It is upper bounded by $\log|G|$; $|C| \rightarrow$ number of changes within a node. It is upper bounded by $|G|$; $d \rightarrow$ the degree of a node; $|W| \rightarrow$ size of an ephemeral subgraph W;$|A| \rightarrow$ size of an ephemeral node; Let $g = \log|G|$.

by allowing a sequence of snapshots with eventlists that record the changes between them. TGI is based on the COPY+LOG idea which, however, is considerably tuned so that it allows a hierarchical structure of snaphsots combined with partitioning of eventlists into small chunks in order to achieve better locality. Furthermore, the system supports lists of different instances of diachronic nodes within the snapshots to facilitate vertex-centric operations. In addition, one of its merits is the dynamic partitioning of the graph. Although in [15], they include partitioning in the qualitative comparison, 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.

G^* is a combination of a vertex-centric approach and the COPY+LOG method, where differences are stored in a compact way by employing redirection. 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).

¿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, therefore, asymptotically space-optimal (see 2nd column). 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 versions of nodes 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 node 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.

3.2 The External Interval Tree

The algorithms and data structures of this work are expressed in the standard two-level external memory model proposed by Aggarwal and Vitter [1]. Our proposed solutions are heavily based on the use of the interval tree data structure [21]. More specifically, let $I = \{(id, [x_i, x_i'])\}, 1 \leq i \leq N$ be a set of N closed intervals on the real line, where id is used as an identifier for each interval $[x_i, x_i']$. We provide a definition for the query type supported by the interval tree.

Definition 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.

Since this work focuses on the external memory, we will use the external version of the interval tree [2]. However, we perform a minor modification as in the following sections we will need to search among all intervals with the same endpoints for a particular interval according to its identifier. In particular, there is a set of lists within each node of the external interval tree that stores in a B-tree intervals based on their left or right endpoint in order to facilitate stabbing queries. Intervals that have same endpoints (left or right endpoint) are consecutively and arbitrarily stored in the B-trees. In our case, we need also to locate a particular interval among them based on a secondary key which we call identifier of the interval. Thus, each time an interval s is stored in a node, if there are other intervals within the B-tree that have the same endpoint then we store s in a sorted order among them based on its identifier. This can be simply supported by either using a B-tree for each such set of intervals with the same endpoints in a B-tree or by adding a secondary key on this B-tree.

THEOREM 1. *Given block size B, we can construct an interval tree on the N intervals of I that uses $O(N/B)$ space and supports interval insertions and deletions in $O(\log_B N)$ time and stabbing queries in $O(\log_B N + K/B)$ time, where K is the number of intervals that overlap the query point.*

Finally, the interval tree can be extended to support open and half-open intervals in a straightforward manner.

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. 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.

4.1 Data Structure Overview

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 fields or 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. It is worth to note that we signify a vertex to be "active" (alive) up until the latest time instance, by setting the t_e value to be $+\infty$. 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 (fields 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 field that has values ℓ_1, ℓ_2, ... (numbers, pointers etc.) during the time interval $[t_s, t_e]$. Note that an edge belonging to v (i.e. one end of the edge is v), can be represented as a field 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 nodes in $\mathcal{I}_v$ maintain the intervals similarly to the nodes of $\mathcal{I}$ discussed in Sec. 3.

Additionally, the diachronic node maintains a B-Tree for each field 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^f_v and is used to maintain the entire history of changes of f between the different snapshots in $\mathcal{G}$. Each record in A^f_v 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 fields taking into account the values ℓ_i, ℓ_j and ℓ_h which were discussed on the previous paragraph. Using the A^f_v trees, which are built on the $[t_s, t_e]$ intervals, we can support stabbing queries for a particular set of fields within the diachronic node, without aggravating asymptotically the space usage. Given the fact that the count of A^f_v trees is dependant on the edge count of v and to facilitate efficient searching of a specific A^f_v tree, we maintain a B-Tree $\mathcal{B}_v$ over all A^f_v 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.

Figure 2 shows the proposed data structure, where $|v_i|$ is the size of v_i (i.e. the count of fields and edges of v_i 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.

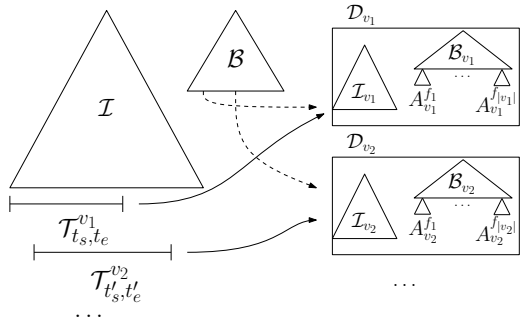

Figure 2: Our proposed data structure

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

4.2 Basic Operations

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

`pointer` p_v `= InsertVertex(id` v`, start time` t_s`, end time` t_e`)` (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.

`values` $\{\ell_1, \ell_2, \ldots\}$ `= ReadAttribute(id` v`, field` f`, time` t`)` (Algorithm 2) returns the set of `values` $\{\ell_1, \ell_2, \ldots\}$ of the `field` f in vertex v at time t[2]. To realize this operation we retrieve the diachronic node $\mathcal{D}_v$ by querying $\mathcal{B}$ for `id` v. Afterwards, we obtain the A^f_v 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.

`void WriteAttribute(id` v`, field` f`, values` $\{\ell_1, \ell_2, \ldots\}$`, start time` t_s`, end time` t_e`)` (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^f_v tree of f for each of the t_s and t_e endpoints. If the stabbing queries do not return any interval

[2]We use a set of values rather than a singular value to make our approach more generic and permit multi-valued attributes

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

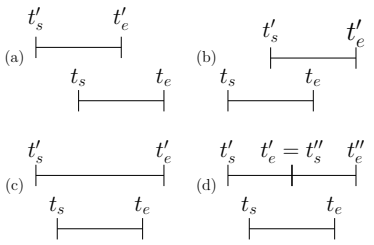

Figure 3: Cases of existing intervals for the field f

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) and follow a different approach. The two cases above are described in detail in the following paragraphs:

1. 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.

2. 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 3). 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 $[t_e, t'_e]$

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**

   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 $[t_e, t''_e]$

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 be deleted and passing a `NULL` value as the set of values.

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.

`pointer` p_u `= ReadVertex(id` v`, time` t`)` (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.

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)

Algorithm 5 SnapshotMaterialization($\mathcal{G}$, t)

Input: evolving graph sequence $\mathcal{G}$, time instance t
Output: a snapshot G_t of the graph at time instance t

1: $V_t \leftarrow \mathcal{I}$.stab($t$)
2: $G_t \leftarrow$ **new** snapshot(t)
3: **for each** *vertex* v **in** V_t **do**
4: G_t.add(v)
5: **end for**
6: **return** G_t

vspace-0.5cm

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 `InsertVertex` 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.

It's worth noting at this point that our indexing module (the $\mathcal{I}$ and $\mathcal{B}$ structures) each maintain a record for each unique vertex (diachronic node) in the sequence. This is in contrast to the indexing module of the original G^* system which maintains a record for each different version of each vertex. This results in a potential time slowdown (since upon locating the diachronic node we also have to retrieve the proper version) but greatly reduces the total space cost. The efficiency of our methods will be further described in Sec.s 5.3 and 6.

Furthermore, 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 is performed in 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 the vertex v at time t (i.e. the count of fields and edges of v at time t).

The following theorem summarizes the results attained in this section.

THEOREM 2. *We can maintain a time-evolving graph sequence in a data structure using optimal $O(m/B)$ 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.

5. APPLICATIONS

In this section, we start by discuss snapshot materialization and graph traversal, which are fundamental application in historical queries, and a case study for local queries, namely graph sampling. Then, we provide an overview of practical considerations during implementation.

5.1 Core Algorithms for Global Queries

Below we explain how to materialize a snapshot 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 3. *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 of all vertices in V_t.*

PROOF. 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 $O(|V_t| \log_B m + \frac{\mathcal{S}}{B})$ (see Algorithm 5). □

THEOREM 4. *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 $O(|V_t| \log_B m + \frac{\mathcal{S}}{B})$ amortized time (I/Os) where $\mathcal{S}$ is the total size of all vertices in V_t.*

PROOF. To perform a depth-first search we require the use of an external stack data structure [22]. The external stack is the external memory equivalent of an internal memory LIFO (Last-In First-Out) data structure and supports

Algorithm 6 DepthFirstSearch($\mathcal{G}$, t, v)

Input: evolving graph sequence $\mathcal{G}$, time instance t, source vertex v
Output: a preordering report of the vertices in G_t

```
1: st ← new external stack
2: D_v ← B.query(v)
3: st.push(D_v)
4: while ¬st.isEmpty() do
5:     D_u ← st.pop()
6:     if D_u is not "visited" then
7:         Mark D_u as "visited" and report it
8:         for each outgoing edge e in ReadVertex(u,t) do
9:             st.push(e.destination)
10:        end for
11:    end if
12: end while
```

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 $O(|V_t| \log_B m + \frac{S}{B})$ time (Theorem 3), which brings the total cost to $O(|V_t| \log_B m + \frac{S}{B})$ worst-case time (see Algorithm 6). □

5.2 Graph Sampling: a local query case-study

Graph sampling [10] is a technique related to picking a subset of vertices and/or edges 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.

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 nodes 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 nodes $\tilde{V}$ is chosen and then E' is the set of all edges that are incident to $\tilde{V}$ while V' is the set of all nodes that are endpoints of the edges in E'. Finally, in TBS, a sampler starts with a set of initial nodes and then extends the sample by following edges from nodes 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 nodes/edges bounded by a predefined budget $\mathcal{B}$ 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 onto it. However, this is contradictory in certain cases since the graph may be so large that we are not able to access it. 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 nodes, which is a main strong point of vertex-centric storage techniques, like ours.

Next, we deal with computing the degree distribution when graph sampling is employed and different storage approaches are followed. VS and Random Walks (RW) have a pretty good performance in approximating the degree distribution of the underlying network (directed or undirected) [24] 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 nodes at the specific snaphsot and then, in the case of RWs, visiting adjacent vertices. TGI [15] would first construct the snapshot at this time instance and move to the sampling process. On the other hand, G^* [18] and our method would only process the nodes that are sampled. This is more efficient when the budget $\mathcal{B}$ is small (e.g., $\leq 10\%$ of the size of the sampled graph). When comparing G^* and our solution, G^* is expected to be more efficient in the case of VS since after discovering the first node we just follow edges which are discovered very efficiently. On the other hand, our solution is more suitable for the VS method, since for each node G^* may need a lot of processing before accessing it when compared to our solution.

Similarly, 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 node by reconstructing the snapshot and then it uses the version chains that connect all versions of a node in a list to speed up the processing. Apart from the obvious problem with the reconstruction of the snapshot when the budget $\mathcal{B}$ is small, the node 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 nodes but because of the fact that all history information of a node 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 node 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 fat node.

5.3 Practical Considerations

The solution we proposed in Sec. 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 $\mathcal{B}$). Furthermore, 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. For practical applications these modifications improve our runtime efficiency at the expense of not strictly following the asymptotic guarantees of Theorem 2, but our approach is still more space-efficient than other proposals. Additionally, the improvements reduce the space cost of our data structure in all cases.

Finally, our solution could be simplified considerably if only transaction time (rather than both transaction and valid time) was considered. We omit details about this discussion due to space constraints.

6. EXPERIMENTAL EVALUATION

In this section, we provide experimental results after incorporating our storage model in the G^* parallel graph processing system. We aim to show (i) actual space savings, and (ii) 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 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 An Overview of the Original G^* System and our extension

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 mostly) fit in the internal memory of each server. While the index is maintained in the main 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 fashion similar to pipelining. The basis of the graph operators is the `vertex` operator that retrieves a particular version of a vertex along with its attributes and edges from the disk.

We incorporate our work into the G^* system by replacing the existing indexing model with the model proposed in Sec. 4 along with the practical improvements described in Subsection 5.3. The system is further modified so that it stores entire diachronic nodes on the 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 version of a vertex.

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.

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

Table 4: Space consumption in real datasets

	Original Indexing Module Size (MB)		Proposed Indexing 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%

The real datasets were obtained from the Large Network Dataset Collection of SNAP [19]. The first dataset is a citation graph of the arXiv hep-th category released as a part of the 2003 KDD Cup [8]. 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 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) [7] or the Barabási-Albert (BA) [3] 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.

6.3 Space Consumption

Table 4 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.

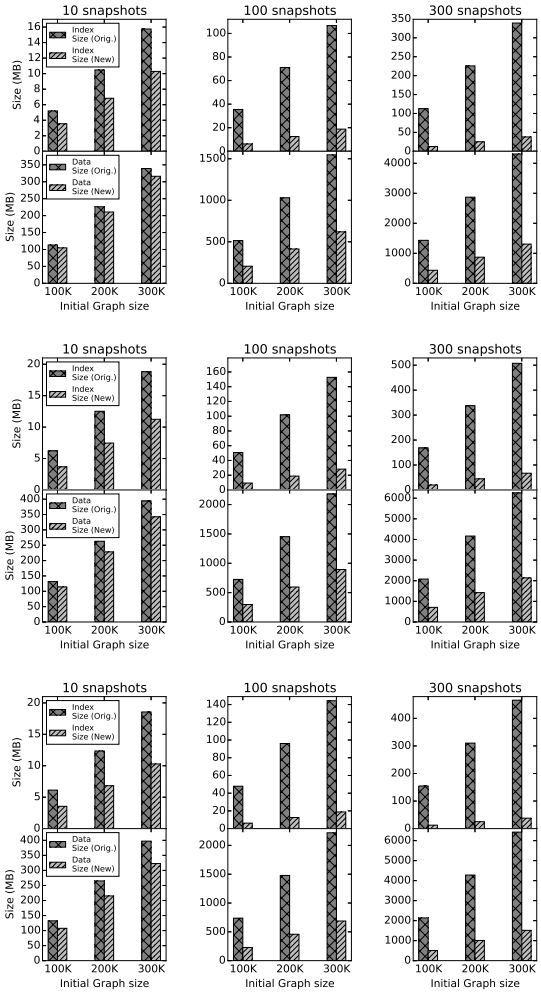

Figure 4: Space Consumption in sequences with $insertion\_rate = update\_rate = 1\%$ (top), $insertion\_rate = 2\%, update\_rate = 1\%$ (middle), and $insertion\_rate = 1\%, update\_rate = 2\%$ (bottom). In each subfigure, the upper part refers to the index size, and the lower to the size of the data files.

Next we experiment with the synthetic datasets for several sequence sizes and insertion/update rates, as shown in Figure 4. Four main observations are as follows. (i) As previously, the space savings reach an order of magnitude. (ii) The higher the insetion/update, the more significant the savings. This can be explained by the fact that, since all vertex and edge updates are stored in the diachronic nodes, the size of the index only becomes larger only when new vertices are created in the sequence. A similar observation can be made about the data file sizes. (iii) In general, our proposed system favors sequences with a higher updates-to-insertions 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 5), where the savings in space there are slightly less, i.e., up to 84% less space.

6.4 Time efficiency for global queries

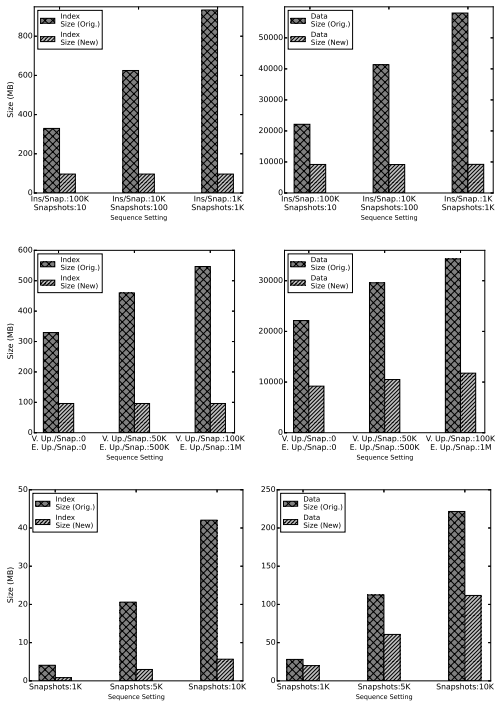

Figure 5: Space efficiency of indices for the BA dataset. Top: effect of granularity (starting vertices $= 2M$ and edges per newly inserted vertex $= 10$). Middle: effect of updates (starting vertices $= 2M$, edges per newly inserted vertex $= 10$, insertions per snapshot $= 100K$, snapshots $= 10$). Bottom: effect of the number of snapshots (starting vertices $= 10K$, edges per newly inserted vertex $= 1$, insertions per snapshot $= 10$).

In the last part of the experiments, we measure the running time for the following global queries, which are in [18]:

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.

The results for the real datasets are reported in Table 5. 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 5 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 versions 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 itself. 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 can achieve significant speedups for the synthetic datasets as well. The results for the ER sequences are shown in Figure 6. For all three types of queries, the maximum reduction in response time is 54%-56%.

Finally, 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 7. 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 8. 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 relative difference remained the same (no figure is provided due to space limitations).

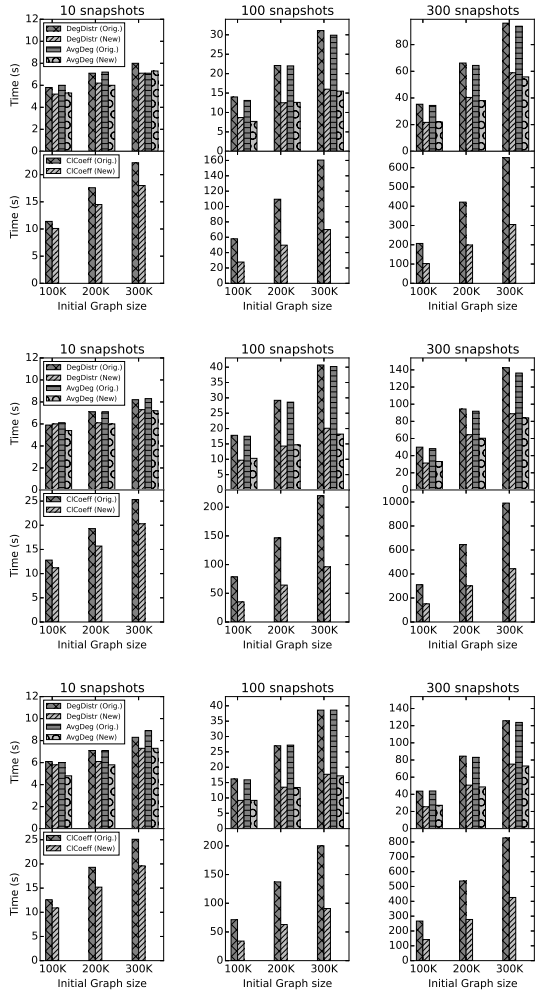

Figure 6: Comparison of time efficiency in sequences with $insertion\_rate = 1\%, update\_rate = 1\%$ (top), $insertion\_rate = 2\%, update\_rate = 1\%$ (middle), and $insertion\_rate = 1\%, update\_rate = 2\%$ (bottom).

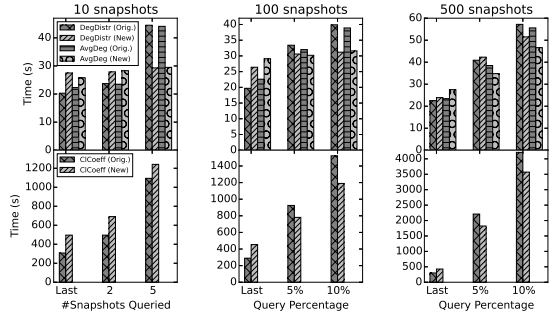

Figure 7: Effect of granularity on time. Starting vertices $= 1M$, edges per newly inserted vertex $= 10$. Insertions/snapshot $= 200K/20K/4K$ for sequences with $10/100/500$ snapshots, respectively.

7. 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 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 savings were up to an order of magnitude, while running times of historical queries dropped to more than a half.

Table 5: Time efficiency in real datasets

	Original G* System Time (s)			Proposed G* 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	-1.90%	-13.77%	18.26%
hep-Ph	9.7	8.4	410	7.7	7.2	471	-20.29%	-14.65%	15.02%
US Patents	329	316	* 213 † 496 ‡ -	242	221	* 234 † 377 ‡ 1204	-26.52%	-30.08%	* 9.69% † -23.84% ‡ -

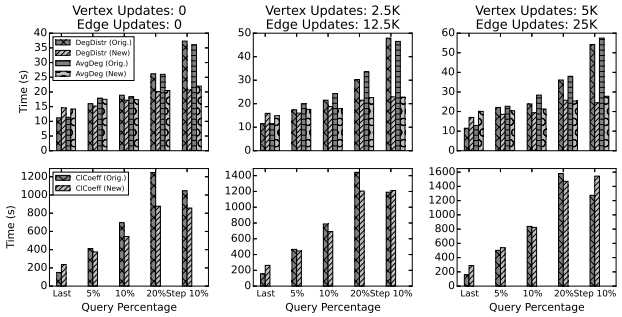

Figure 8: Effect of updates on time. Starting vertices = $1M$, edges per newly inserted vertex = 5, snapshots = 10.

8. 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] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.

[3] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[4] C. "big graph" processing library. `https://github.com/twitter/cassovary`.

[5] 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.

[6] 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.

[7] P. Erdős and A. Rényi. On random graphs. I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[8] J. Gehrke, P. Ginsparg, and J. M. Kleinberg. Overview of the 2003 KDD cup. *SIGKDD Explorations*, 5(2):149–151, 2003.

[9] A. Giraph. `http://giraph.apache.org/`.

[10] P. Hu and W. C. Lau. A survey and taxonomy of graph sampling. *CoRR*, abs/1308.5865, 2013.

[11] W. Huo and V. J. Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *SSDBM*, pages 38:1–38:4, 2014.

[12] 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.

[13] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: mining peta-scale graphs. *Knowledge and Information Systems*, 27(2):303–325, 2011.

[14] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, pages 997–1008, 2013.

[15] U. Khurana and A. Deshpande. Storing and analyzing historical graph data at scale. In *EDBT*, pages 77–88, 2016.

[16] G. Koloniari, D. Souravlias, and E. Pitoura. On graph deltas for historical queries. *WOSS*, 2012.

[17] 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.

[18] 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.

[19] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[20] 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.

[21] E. M. McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles, 1980.

[22] R. Pagh. Basic external memory data structures. In *Algorithms for Memory Hierarchies*, pages 14–35, 2002.

[23] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. *PVLDB*, 4(11):726–737, 2011.

[24] B. F. Ribeiro and D. Towsley. On the estimation accuracy of degree distributions from graph sampling. In *CDC*, pages 5240–5247, 2012.

[25] K. Semertzidis, E. Pitoura, and K. Lillis. Timereach: Historical reachability queries on evolving graphs. In *EDBT*, pages 121–132, 2015.

[26] B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD*, pages 505–516, 2013.

[27] 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.

[28] 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.