Scalable Time-Versioning Support for Property Graph Databases

Warut D. Vijitbenjaronk[†], Jinho Lee[†1], Toyotaro Suzumura[*], and Gabriel Tanase[*2]

IBM T.J. Watson Research Center
[†]Austin, TX [*]Yorktown Heights, NY
warut.vijitbenjaronk@ibm.com, {leejinho, suzumura, igtanase}@us.ibm.com

***Abstract*—When graphs change over time, it is important to make the changes trackable for many graph-based applications. We propose an implementation of OLTP-oriented graph database that supports time-versioning. There has been a few snapshot-based approaches for supporting time-versions, but they usually require the full-restoration of the graph, and lack the resolution of the time space. Using a B-tree as the datastructure for the backend storage, our database allow fast and scalable support for restoring the arbitrary part of the graph, without slowing down the normal accesses to the current graph. Experimental results show that our scheme is much efficient than the straightforward solutions, in terms of space and performance.**

***Keywords*-graph database, time versioning, OLTP**

I. Introduction

Graphs evolve over time, and we need to keep track of them. While there are some stationary graphs, most graphs we find from the real world get changed over time. For example, in social network graphs, there are new users every moment, and relationship between users are frequently updated. Also, new links are being formed between web pages for webgraphs [19] and new papers are added to the citation networks.

To store and analyze these graph data, numerous models, frameworks, and databases has been proposed. However, there are not many OLTP-type graph database that supports time-versioning. In fact, most of the popular graph databases [20], [2] do not support such feature, and many people try to find a work-around as in [5]. However, such workarounds would be inefficient and unscalable when the length of history becomes longer.

There is some work on supporting snapshots, usually with *CSR*s (Compressed Sparse Row) which favors OLAP style analysis [11], [17]. Snapshot based approaches usually set a few checkpoints, and provide methods to go back to the specified checkpoint. While leaving a complete copy is at one extreme, many approaches provide deltas between snapshots [12], or use a hybrid between them [17]. However, the snapshot approaches have some fundamental limitations. First, it forces the quantization of the time space. It not only gives the problem of determining the range of the time for a single snapshot, but also the unavailability for accessing a time in the middle of a snapshot. Also, snapshots often require the restoration of the whole graph, even if one wants to access a small portion of the graph, which can be quite costly for large graphs.

In this paper, we present an implementation of a graph database that supports time-versioning for property graphs. To the extent of our knowledge, this is the first paper that discloses the detailed design of OLTP style graph database that supports time versioning. Exploiting the B-tree structure of the backend storage, our design provides graceful scaling of the access times with the growth of the graph and its history. From our observation that only a part of graph usually accessed with a certain timepoint, we keep a separate history for each entity (i.e., vertex, edge, and property). Also, our design supports fine-grained timed access with its resolution as high as the resolution of the data type that is used by the timestamp. We provide an evaluation of the proposed scheme against a few existing solutions on a state-of-the art server hardware.

The contribution of this paper can be described as follows:

- We describe in detail the internal design of the graph database that supports time versioning.
- The proposed design supports fine-grained timepoint for versioning.
- Through the use of B-tree structure, an entity at an arbitrary timepoint can be obtained in a single lookup, without having to restore the whole graph.
- We further discuss a few possibilities for optimizing the performance of the design.

II. Related work

In recent years, many advancements have been made in graph processing and graph storage to address inherently interconnected data while focusing on different aspects and in a variety of contexts including social networks, biological interactions, and financial transactions.

A. Graph Databases

These graph processing systems can be separated into two major paradigms: OLAP (online analytical processing) and OLTP (online transaction processing). OLAP models are

[1]Corresponding author
[2]The author is now with Graphen Inc.

generally optimized for performing computations on large, predominantly fixed graphs, while the latter are designed to handle dynamically evolving graphs. Within the former category, Pregel [18] and GraphLab [16] are one of the first of these models. They provide a powerful tool for OLAP analysis by performing highly parallel computations using a vertex-centric model. There are also out-of-memory graphs processing frameworks. GraphChi [13] has shown that finely optimized out-of-memory processing could be more efficient than in-memory approaches with a cluster of machines. It has led to the surgence of many successors [21], [23], [14]. OLTP systems include Neo4J [20], AllegroGraph [1], and JanusGraph [2]. Although some OLTP graph databases do not natively support temporal logical queries, it is possible to implement this functionality at the cost of significant space and performance overheads by storing timestamps as properties of vertices, edges, or both [5].

B. Time-Versioning Support

Although not extensively studied in the context of graph processing, the problem of providing a time-version support for databases is one of the classic problems, and there have been a number of literature in this area. For example, [22] provides a thorough review of the methods for storing and querying the timed data.

Two conceptual methods for performing time-versioned queries on transaction time databases are suggested by Salzberg and Tsotras [22], known as the *Copy* and *Log* methods. The *Copy* method maintains a linear series of snapshots, and thus can quickly perform queries at any point in time where a snapshot has been created. However, it stores multiple copies of data, and is prohibitively expensive in terms of storage. In contrast, the *Log* method maintains a list of changes performed on the database. It is space-efficient, but requires an expensive traversal of the log in order to recover the state of the data. The *Copy+Log* family of methods combines a finite set of snapshots with a list of deltas between them.

Recently, there has been some attention on time-versioned graph databases. One of those is DeltaGraph [11], which uses a hierarchical collection of snapshots separated by deltas. This is a hybridization of the aforementioned *Copy* and *Log* methods, and reduces both the space and computational footprint. In contrast, the LLAMA framework [17] attempts to reduce the memory footprint of snapshots by storing only the deltas between snapshots explicitly, and a pointer to a previous snapshot for stationary portions of the graph. Chronos [9] provides speedup methods for graph processing by exploiting the snapshot-level parallelism. Also, GraphTau [10] suggests a two computational models that can take advantage of the evolving graph data.

In all of these approaches, the temporal granularity of the database is limited to the frequency of the snapshots. Especially in domains where fine-grained adds and reads are vital, it is important to be able to simulate continuous time and to perform queries at arbitrary points in time, as well as transient structures, which we define as those which exist at exactly one instant in time.

Table	Key	Value
ex2i	ex_id	vid
i2ex	vid	ex_id
vi2e	src, tgt	eid
vi2p	tgt, src	eid
v/e_property	v/eid, pid	pvalue

Table I
THE BASELINE SYSTEMG STRUCTURE

III. BASELINE GRAPH DATABASE

A graph is usually comprised of vertices, edges, and their properties, in which we use the term *entity* to represent any one among them. As a baseline graph database, we use IBM SystemG [4] distributed graph database. As it backend storage, SystemG uses LMDB [3], a B-tree based key-value store. Because of its internal B-tree structure, it becomes an excellent platform for implementing a time-versioned graph.

Table I shows the organization of the LMDB databases in SystemG. SystemG assumes that the vertices are referenced by a unique external id in a string form which is usually human-recognizable. Internally, each vertex is assigned a unique internal id in an integer form, which is often much efficient than handling a string. These internal ids are used in reference to the vertices for source and targets of edges, or the owner of properties.

The existence of a vertex is stored as a external-to-internal map (*ex2i*) and the internal-to-external map (*i2ex*). When a new vertex is added, an internal atomic counter assigns an id to the vertex, and the one entry is added to each of the ex2i and i2ex tables.

The edges are identified by their source and target, and multiple edges between pairs of vertices are supported. The SystemG API provides support for both directed and undirected graphs. The LMDB outgoing (*vi2e*) edge table is indexed by the source vertex. The value side maintains the target vertex id, optionally a user-assigned label, and the unique edge id, which is maintained similarly to the vertex id. In case of a directed graph, SystemG keeps another database instance (*vi2p*) for incoming edges. When adding an edge, the user provides the external ids of the source and target. Both the source and target vertices are checked on the *ex2i* table for their existences. The vertex id are retrieved, or created if not existing. Then an entry is added to the edge table with an assigned edge id.

There are separate tables for storing vertex/edge properties, which are organized as shown in *v/e_property*. The properties are indexed by the corresponding vid or eid, followed by the property id, unique for each kind of property. Each vertex and edge can have multiple properties.

IV. Design Overview

A. Timed APIs

As a simple extension to the original APIs, a timestamp is added to the methods that writes to the database (add/delete/modify). For example, "add an edge from A to B" in a normal graph database is changed to "an edge is added from A to B at time t". The 'history' of the writes made to the entities are stored with the corresponding timestamp, so that the graph at a past state can be obtained. For reading methods (get), a timestamp is also added for indicating the time point that the graph has to be restored to. A time range queries are also supported with two timestamps for the start and the end of the range, which answers the question "What are the entities that were valid during the time range". However, we do not discuss them in detail since extension from a single-timed get method to a range-timed get method is straightforward. For convenience, we provide a *set_current_time()* method for a graph handle. When the '*current time*' is set, all method is assumed to receive the current time of the graph instance as the implied timestamp. For the details of the provided APIs, please refer to Section VI.

To maintain the of validness of the history, we introduce "advance-only within an entity" policy to the event time of the write methods. When changes are made to the middle of a history, its meaning is often ambiguous. Therefore we only allow changes with timestamps after the point where the latest event happened.

Such model is close to a transaction-time model [8], but with more relaxed constraints. In the transaction-time model, only the actual time that the transaction has been made is recorded to each entry, and no modifications to the past, or future is allowed. However in the model used in this paper, the 'advance-only' has to be kept only in the context of each entity. We do provide a set of low-level methods to put modification history of the entities without any restriction, but it is the user's responsibility to manually make the entity's history logically meaningful.

B. Design Philosophy

For our implementation of time-versioning support, we consider a number of criteria that conform to the most common types of queries. Figure 1 shows the rough overview of the design. The design choices we made can be summarize into three following items: isolated history entries per each entity type (vertex, edge, and property), separation of the current graph from the stale history, and B-tree based history table searchable by timestamps. In the rest of the subsection, we describe these choices in more detail with their justifications.

In many cases, OLTP databases are latency-bound. Especially when the graph database is used as a backend of a real-time service such as a financial application, the graph

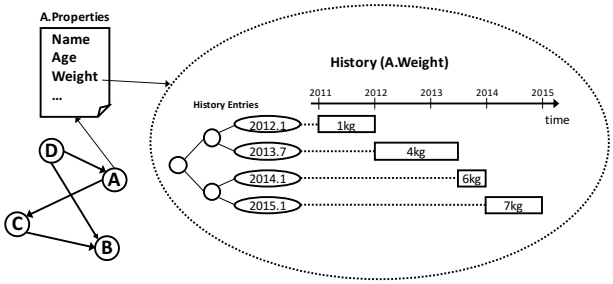

Figure 1. Overview of the time-versioned graph database.

database usually has a time budget of a few milliseconds. As a result, most of the queries to the OLTP oriented graph databases are relatively small, and often egograph-based with a few hop distances. For this reason, restoration of the full graph on every timestamped access would result in a waste. In the proposed graph database, the history tables are clustered with regard to each entry, so that only the entities of interest can be restored.

Another important feature is to keep a separate 'current' database table for the most recent valid data. In the setting of a social network, for example, current users and recent posts are more likely to be referenced than older data. As such, the separate current table attempts to minimize slowdowns to read access times for current data, due to the introduction of time versions. When the entry in the current graph is not valid for the queried timestamp, separated tables for history are read for older versions.

To make the access time of the graph database scalable to the size of the history, we place a timestamp of the history entry as the last entry of the key side of the backend KV store. Since we use a B-tree data structure for the KV store, the entries are first ordered by entity identifiers (e.g., vertex id) and then the timestamps as the last radix. Thus, when reading an entity at certain timepoint, a single lookup on the tree would yield the entry which has the smallest timestamp greater than the timestamp of interest.

Other than the proposed method, one alternative approach is to add timestamps as properties of entities, as exemplified in [5]. In addition to the existing properties, a few properties can be added as "create", or "deleted", and graph queries can be made against specific conditions on those properties. This method has a strength in that it can be done on top of almost any kind of property graph models [20], and it does not require changes to the graph database implementation. However, as shown in [5], it would make the queries very complicated. Furthermore, as the history becomes long, it would slow down the query processing time.

Another method is to chain the past history of an entity to its most recent version as a linked list, as done in [7]. This is better than the approach above since it wouldn't hurt the access time of entities with short histories. However,

Table	Key	Value
ex2i	ex_id	vid, *birth, death*
i2ex	vid	ex_id
v_history	*vid, death*	*birth*
vi2e	src, tgt	eid, *birth*
vi2p	tgt, src	eid, *birth*
vi2e_history	*src, tgt, death*	*eid, birth*
vi2p_history	*tgt, src, death*	*eid, birth*
v/e_property	v/eid, pid	pvalue, *write_time*
v/e_prop_history	*v/eid, pid, overwritten_time*	*pvalue*

Table II
THE MULTIVERSIONED SYSTEMG STRUCTURE

the linked list structure give a long access time when the query access the deep history. The access time would be proportional to the length of the history for the specific entity in the query and thus unscalable.

V. IMPLEMENTATION DETAILS

In this section, we describe the changes to the graph database tables we made to support time-versioning of the data. We use LMDB [3] as the backend. However, any backend storage that uses a variant of tree-based structures could be used instead. Table II displays the structure of the keys and values of the backend KV store. The newly added items are emphasized in italic.

For vertices (*ex2i*) table, two timestamps are added to the vertex ids: One for creation time, another for deletion time. The creation time is written when the vertex is added, and the deletion time is written on its deletion. In principle, these timestamps are read-only and never overwritten. Together, the pair of timestamps form a 'valid' interval, and on reading the vertex, the interval can be compared with the timestamp in search to figure out whether they overlap or not. However, while the pair of timestamps yields the current state of the graph, it is an incomplete record of deeper history, such as a vertex being created and deleted repeatedly.

Therefore, for the purpose of keeping the deep history, an additional table for vertex history (*v_history*)is added to the graph database. Instead of using external id as the key, we chose the internal id as the key of the vertex history table. At the moment when the history table is accessed, the ex2i table should have been already accessed a priori so that the internal id should be already available. In addition, one timestamp for its deletion time is appended to the vertex id as the key. By doing this, when searching for a history of a vertex from a certain time point, the desired history entry can be found in a single search to the backend storage, with the logarithmic time complexity provided by the B-tree structure. On the value side, the other timestamp (creation time) will be stored to complete the valid interval.

While the whole history can also be embedded into the original vertex table, it would increase the size of the table, and would cause adversarial effects to the queries to the 'current' graph, especially when the size of history gets larger. Thus, to optimize the performance for accessing the current graph, but still support the versioning, we decided to keep a separate table for history entries.

We do not modify the i2ex table from the original graph database. When the vertex id is available, it usually implies that the timestamps associated to the vertex is already available, so embedding the timestamps again to the i2ex tables would be redundant.

Similar, but different changes are made to the edge tables (*vi2e* and *vi2p*). The most difference in edges is that multiple edges are allowed between a pair of vertices. As a side effect, a deleted edge never gets created again, unlike the vertices. Thus, in the edge table, we store only the alive edges, and no deletion time is marked with them, since they are all alive. In the history table (*vi2e/p_history*), the source vertex id, target vertex id and the deletion time are used as the key.

For properties and their history entries, we store only one timestamp for the modification time, which is functionally identical to the creation time for vertices or edges. Therefore, the valid interval of a property is from its own modification time to the modification time of its successor. We decided that the deletion of a property would have little meaning to the databases (e.g., deleting the "name" field from a user), and we only allow modifying the value of the property. When it is really needed to delete a property, we replace the property value to be a special "invalid" marker. If a property does not exist in certain entity, it is considered invalid as well. When a new kind of property that did not previously exist is introduced to the graph database, its creation time is regarded as the creation time of the entity that the property belongs to, and its value is assumed to be "invalid" until it is set to a different value. In the history table (*v/e_prop_history*), the vertex/edge id, property id, and the modification time will be used as the key, and the property value is placed at the value side.

VI. ALGORITHMS

In this section, we describe the basic algorithms for a few graph database APIs, which will be the building blocks for other more complicated algorithms. Please note that all the algorithms shown in this section are pseudo-code, and many details are omitted.

Below shows the algorithm for adding a vertex.

```
1  addVertex (G, exid, time)
2    if (time!=invalid) time=G.cur_time
3    if exid in G.ex2i:
4      (vid, ct, dt) = G.ex2i.get(exid)
5      if (dt > time) return err;
6        G.ex2i_hist.set(
7            key=(vid,dt),value=(ct))
8    else: //new vertex
9      vid = G.new_vid()
10     G.i2ex.set(key=vid,value=exid)
11   G.ex2i.set(key=exid,
12           value=(vid, time, inf))
13   return vid
```

As explained in Section IV-A, each method is assumed to take a time parameter. However, for convenience, a graph instance holds a variable for "current time". When the methods are called without a timestamp, the "current time" stored in the graph instance is used instead (line 2). For adding a vertex, the ex2i table is first checked to see if it already exists, or has existed in the past. if an entry is found, its timestamps are checked. If the deleted time is later than the timestamp provided, then it is the violation of the rule discussed in Section IV-A and the function just returns with and err (line 5). otherwise, the existing entry is moved to the vertex history table (line 6-7) and a new entry with the timestamp as the created time is added to the i2ex table (line 11-12). If no match is found for the exid provided in ex2i table, it's a completely new vertex, and a new vertex id is allocated (line 9), and it is added to i2ex table (line 10) as well as to the ex2i table. Lastly, the method returns with the vid for the added vertex.

```
14 getVertex (G, exid, time)
15   if (time!=invalid) time=G.cur_time
16   if exid in G.ex2i:
17     (vid,ct,dt) = G.ex2i.get(exid)
18     if (time ∈ [ct,dt]) return vid
19     ((vid_hist,dt), ct)
20         = G.ex2i_hist.get((vid,time))
21     if vid==vid_hist
22         and time ∈ [ct,dt]:
23       return vid
24   return none
```

getVertex() method works similar to adding a vertex. The ex2i table is checked for the existence of exid (line 16). If it is not found, it does not exist in the graph. If an entry is found, its valid interval is checked against the search time parameter. If the search time is within the valid interval, the vertex is valid, and the vid is returned (line 18). if is does not match, the history table is searched for the vid with the search time (line 19-20). When exact match for the key (vid,time) is not found (which is likely in most cases), the table returns the next largest key with its associated value. Therefore, when there is a matching vid with the valid interval containing the search time, it will be returned in a single lookup.

```
25 delVertex (G, exid, time)
26   if (time!=invalid) time=G.cur_time
27   if exid in G.ex2i:
28     (vid,ct,dt) = G.ex2i.get(exid)
29     if (dt != inf) return err
30     G.ex2i.set(key=exid,
31         value=(vid,ct,time))
32     G.delete_edges_from_to(vid)
33   else return err
```

For deleting a vertex, the current entry in the ex2i table is retrieved (line 28) and its deletion time is replaced with the deletion timestamp (line 31). Because the vertex has been deleted, all the edges that are outgoing or incoming to/from the deleted vertex have to be deleted together (line 32). This is done similar to delEdge method (line 57-68), but using only the src for vi2e table and tgt for the vi2p table.

For adding adges, an edge instance is added without checking for an already existing one, because multiple edges are allowed between a pair of vertices. The algorithm is shown below.

```
34 addEdge (G, src, tgt, time)
35   if (time!=invalid) time=G.cur_time
36   sid=G.addVertex(src,time)
37   tid=G.addVertex(tgt,time)
38   eid=G.new_eid()
39   G.vi2e.add(key=(sid,tid),
40       value=(eid,time,inf))
41   G.vi2p.add(key=(tid,sid),
42       value=(eid,time,inf))
43   return eid
```

First, it is checked whether the source and target vertices exist in the graph (line 36-37) because it is a convention for many graph databases to disallow dangling edges. The addVertex method will add the vertex if needed, and the id of the newly added or already existing vertex will be returned. With the obtained vertex ids, and edge id is generated (line 38) and it is added to the vi2e table for outgoing edges, and vi2p table for incoming edges with the provided timestamp.

There are also many forms of functions for retrieving edges due to the multiple edges between a pair of vertices. Below is an example function for retrieving all edges between a pair of vertices.

```
44 getEdge (G, src, tgt, time)
45   if (time!=invalid) time=G.cur_time
46   sid=G.getVertex(src,time)
47   tid=G.getVertex(tgt,time)
48   if time==G.cur_time:
49     elist=G.vi2e.getall(sid,tid)
50   else elist=G.vi2e_hist.getall(
51                   sid,tid,time)
52   ret=list()
53   for (eid,ct,dt) in elist:
54     if time ∈ [ct,dt]:
55       ret.add(eid)
56   return ret
```

After finding the vertex ids (line 46-47), the vi2e table is searched for the source and the target if the method is looking for the currently valid edges. In this case, the edges obtained are assumed to have the deletion time of infinite. If the search time is not the current time, the edge history table is searched for the edges with provided source and target, and the deletion time later than the provided search time. The edges found are checked for the search time, and those which have interval containing the search time are returned.

Similarly, deleting edges can take multiple forms, such as deleting all edges between a pair of vertices. In the code below, we provide the pseudo-code for deleting an edge based on its edge id.

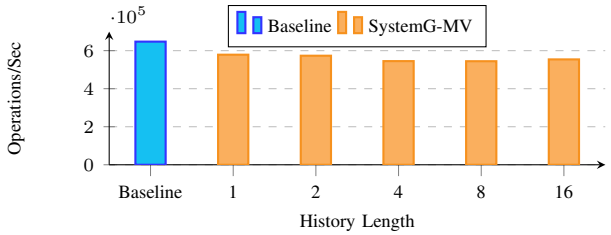

Figure 2. Performance of reading current vertices.

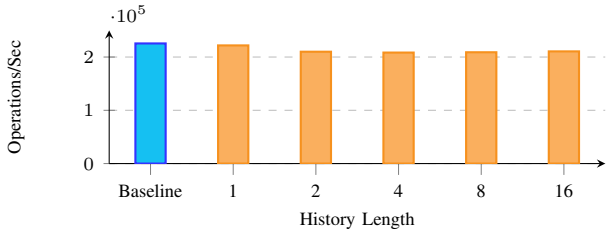

Figure 3. Performance of reading current edges.

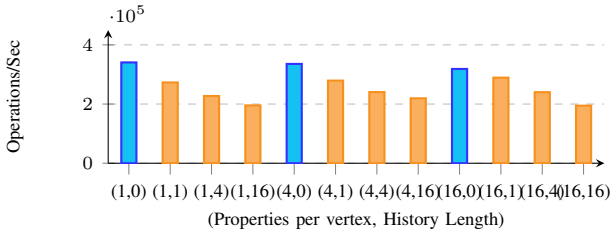

Figure 4. Performance of reading current vertex properties.

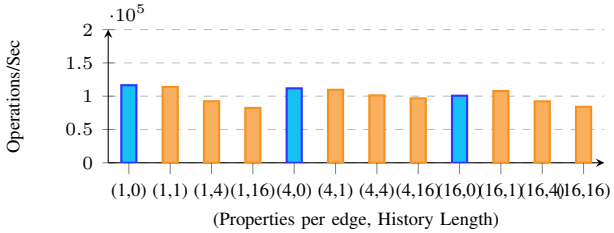

Figure 5. Performance of reading current edge properties.

```
57  delEdge (G, src, tgt, eid, time)
58    if (time!=invalid) time=G.cur_time
59    elist=G.vi2e.getall(src,tgt)
60    for (eid_cand,ct,dt) in elist:
61      if eid==eid_cand
62          and time > ct and dt==inf:
63        G.vi2e.remove(src,tgt,eid_cand)
64        G.vi2p.remove(tgt,src,eid_cand)
65        G.vi2e_hist.set(key=(src,tgt,dt),
66              value=(eid,ct))
67        G.vi2p_hist.set(key=(tgt,src,dt),
68              value=(eid,ct))
```

In line 59, all the entries for edges that connects src to tgt are retrieved, and there are checked against the eid to be deleted (line line 61). Upon finding the entry, it is removed from both the vi2e and vi2p table, and moved to the history tables (line 65-68).

For properties, we provide two methods set and get, since we do not allow deletion of properties. There are separate methods and tables for vertex properties and edges properties, but we display only those for vertices, since the algorithms are identical.

```
69  setVProp (G, vid, pname, pvalue, time)
70    if (time!=invalid) time=G.cur_time
71    pid = G.get_pid(pname)
72    if (vid,pid) in G.v_prop:
73      (old_pv,wt)=G.v_prop.get(vid,pid)
74      if wt > time:
75        G.v_prop_hist.set(
76                  key=(vid,pid,time),
77                  value=(old_pv))
78    G.v_prop.set(key=(vid,pid),
79          value=(pvalue,time))
```

For setting properties, a property id is read or newly allocated from the property name (line 71). Then the vid, pid tuple is searched for the property table (line 73). The found entry is moved to the history table, with its overwritten time as the provided write time (line 75-77). Lastly, the new entry is added, or overwritten to the property table with the new value and the write time.

```
80  getVProp (G, vid, pname, time)
81    if (time!=invalid) time=G.cur_time
82    pid = G.get_pid(pname)
83    (pv,wt)=G.v_prop.get(vid,pid)
84    if (wt < time) return pv
85    ((vid_cand,pid_cand,ot),pv) =
86          G.v_prop_hist.get(vid,pid,time)
87    if vid==vid_cand and pid==pid_cand:
88      return pv
```

Above shows the algorithm for getting vertex property. First, the property table is read with the vertex id-property id tuple, and its written time is checked (line 83-84). If the written time is later than the search time, the history table has to be looked up with the vid, pid and the search time (line 85-86).

VII. Evaluation

Our evaluation focuses primarily on the performance of the versioned graph in comparison to the baseline non-versioned implementation discussed in our earlier paper [4] and against an alternative model based on the reverse chaining method. In the legend, they are named 'SystemG-MV', 'Baseline', and 'Reverse Chaining', respectively. We evaluated the performance of these three systems on the following criteria: vertex modification/retrieval, edge modification/retrieval, property modification/retrieval, and local breadth-first search (BFS) queries. The goal of each of these experiments is to evaluate the overhead resulting from supporting versioned queries and the system's performance on common operations in an OLTP environment, namely data ingestion and read-only queries.

A. Setup

The experiments were performed using an IBM S824L machine. The machine features 4 NUMA nodes, 24

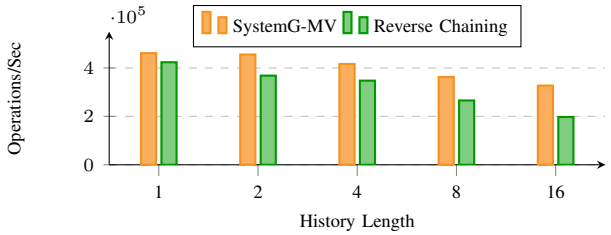

Figure 6. Performance of reading historic vertices.

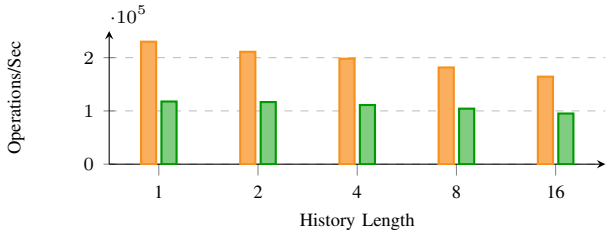

Figure 7. Performance of reading historic edges.

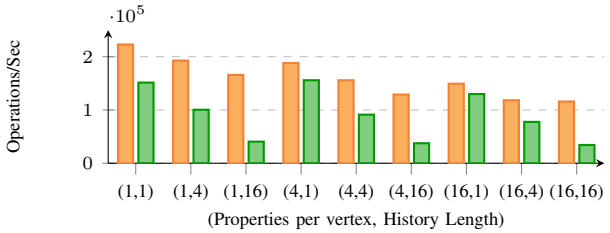

Figure 8. Performance of reading historic vertex properties.

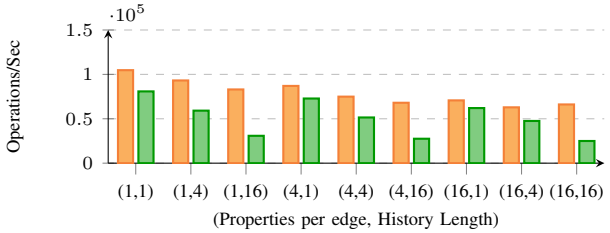

Figure 9. Performance of reading historic edge properties.

POWER8E cores with SMT8 at 3.3GHz, 2TB memory. It is also connected to an IBM FlashSystem 840 Enterprise SSD storage for loading input data files.

To mimic the evolving graphs from the real world, we adopted the forestfire model [15] to generate a synthetic growing-only graph with the size of 100,000 vertices and 1.25 million edges when fully grown. On top of the graph, we randomly modified the vertices, edges, and properties to leave the history at various size of the history events to capture the scalability with respect to the history size. We define the history size as the ratio between the currently alive (or valid) entities and the dead (or stale) entities. We swept the size of history from 1 to 16. Also, different number of properties per entity has been tested together.

B. Data Querying

We assess the performance of each system with retrieving edges, vertices, and properties. We divide the experiments into two sets. In the first set, the baseline and the proposed versioned database are compared on querying the currently alive entities of the graph. In the second set, the graphs are queried at arbitrary timestamps, on the proposed multiversioned database and the reverse chaining. The entries queried in this experiment are chosen at random from the dataset.

Figure 2 and Figure 3 display the performance for reading vertices and edges, respectively. Since our implementation optimizes towards reading current graph by maintaining separate table for current graphs, the performance is almost the same for baseline and the SystemG-MV. Also, the change in the history size does not impact the performance for reading the current status of the graph. Performances for reading the vertex/edge properties in Figure 4 and Figure 5 show similar trend with marginal differences.

The performance of reading past graph give more interesting insights. Figure 6 shows the performance of vertex read queries for SystemG-MV and Reverse Chaining. As the size of history gets larger, the performed operations per second goes down for both implementations. However, the slope of degradation is much steeper in reverse chaining. In reverse chaining, the linked list for each entity has to be chased until the entry valid at the queried time is found, leading to extra overhead, especially for queries with long histories. However, SystemG-MV can directly search for the entry with the wanted timestamp at a single lookup. Thus the slowdown is much shorter, and the query time does not depend on which time point the query is trying to access, unlike the reverse chaining. In Figure 7, similar data points for edge read queries are shown. The key difference with vertex queries are that the rate of slowdown is not as high as in the edge read queries. Because there can be multiple edges between a single pair of vertices, a once-dead edge never gets revived, and therefore no chain can be formed. As a result, similar algorithms are used for edge queries in SystemG-MV and reverse chaining, and the slowdown for both is due to the growth of the B-tree in the LMDB table.

Figure 8 and Figure 9 show almost identical curve shape. Similar to the vertex queries, SystemG-MV shows log-scale slowdown on history size increases while reverse chaining experiences a linear slowdown. Thus the speedup difference is much larger when the history length is larger. The number of properties, on the other hand, causes similar slowdown effect to both multiversion implementations, coming from the LMDB table growth.

C. Data Ingestion

We also performed data ingestion separately for vertices, edges, and properties using a synthetic dataset. For this experiment, we compare the proposed versioned system to a non-versioned baseline and to the reverse chaining model suggested earlier.

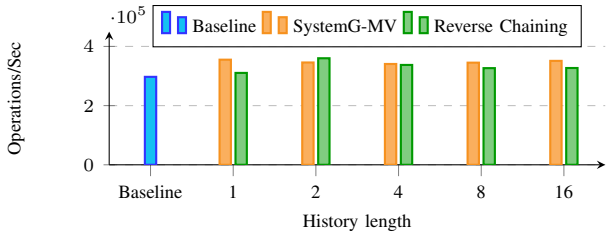

Figure 10. Performance of adding/deleting vertices.

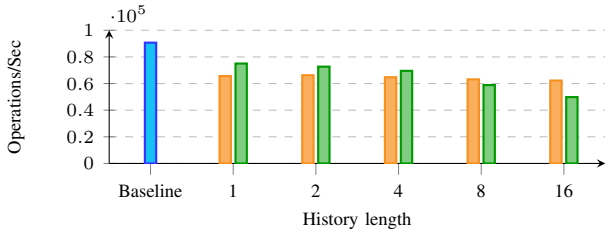

Figure 11. Performance of adding/deleting edges.

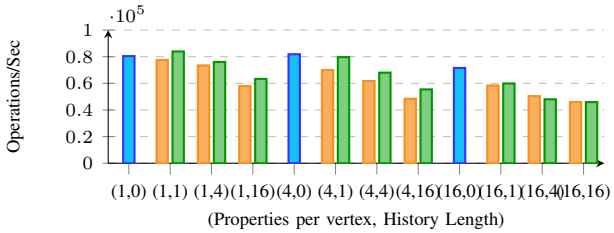

Figure 12. Performance of setting vertex properties.

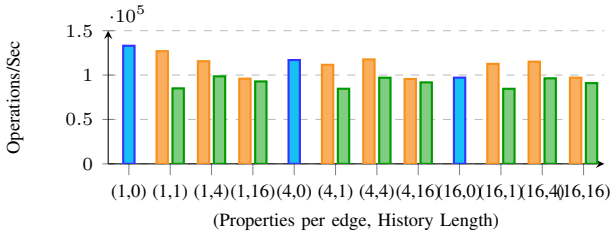

Figure 13. Performance of setting edge properties.

In order to perform vertex ingestion, we simulate a series of vertex additions and deletions at progressively increasing timestamps, referred to by external IDs. Figure 10 shows the performance results. We found that as a result of the vertex insertion and deletion algorithms for all three models, the speed is bound by the LMDB write performance, and that there is no significant performance overhead for the multiversioned systems when compared to the baseline. As a matter of fact, the baseline pays more overhead at deleting vertices, sine all the corresponding properties has to be deleted in the baseline, where the both multiversioned implementations keep the properties for accessing history.

Experiments for edge ingestion are similarly done. We performed edge insertions and deletions at progressively increasing timestamps, referred to by the external IDs of the source and target. The performance is shown in Figure 11. The performance for edge ingestion remains fairly constant regardless of the number of existing edges between pairs of vertices. Due to the growing data size as a result of storing historical data, the time overhead compared to the baseline increases in a log scale with the size of the history.

Property ingestion performs edge and vertex property updates on a graph of 1.25 million edges, with up to 16 different properties on each. The results are shown in Figure 12 and Figure 13. Similar to the vertex and edge ingestion tests, this test simulates serialized transactions. In property ingestion, the two versioned graph databases have to access the two tables for the current properties and the past properties. Also, there is an unavoidable linear growth in storage proportional to the number of properties and the average history size. Sometimes, reverse chaining performs slightly better than the SystemG-MV on ingestions. This is because reverse chaining places the new entry at the head of the its chain, and does not pay its usual overhead of chasing the chain. Also, a little speedup accounts for having smaller keys due to having timestamps in the value side. At expense, reverse chaining suffers from long read times for accessing historic entities as shown in the previous subsection.

D. BFS

Breadth-first search was performed on the same dataset, with various length of histories and properties per entities. Similar to the previous subsection, we perform BFS once on the current state of the graph comparing the baseline with the SystemG-MV, and again at the past state, comparing SystemG-MV with the reverse chaining. We constrain the maximum depth of the query to three levels, and a property is accessed per every visited vertex. We perform 10,000 sequential queries with different sources, and measure performance as the average number of queries per second. The results are shown in Figure 14 for current graphs, and Figure 15 for graphs at past state.

Results indicate that there is an average 7.7% loss in performance for queries on the current state of the versioned graph when compared against the baseline model for all graph sizes. For queries on a historical state of the database, we find that there is a significant improvement in performance for the SystemG-MV compared to reverse chaining, which widens as both the number of properties and the size of the history increases. As shown in Figure 15, the improvement ranges from 38.5% for the smallest graph to 253.4% for the largest one. In the scenario where a user expects that multiple queries will be performed at a specific timestamp, the cost of performing traversals can be reduced significantly by implementing the early restoration policy discussed in VIII-B.

VIII. Discussion

A. Time Complexity Analysis

Table III displays comparison of the time complexity of the selected methods from Section VI, for baseline non-versioned, versioned with current time, and versioned with

Method	Baseline	Timed (current)	Timed (past)
$addVertex$	$O\big(2log(\|V\|)\big)$	$O\big(2log(\|V\|)\big)$	$O\big(2log\|V\| + log(\|V\| \cdot h)\big)$
$getVertex$	$O\big(log(\|V\|)\big)$	$O\big(log(\|V\|)\big)$	$O\big(log\|V\| + log(\|V\| \cdot h)\big)$
$addEdge$	$O\big(4log(\|V\|) + 2log(\|E\|)\big)$	$O\big(4log(\|V\|) + 2log(\|E\|)\big)$	N/A
$getEdge$	$O\big(2log(\|V\|) + log(\|E\|) + r\big)$	$O\big(2log(\|V\|) + log(\|E\|) + r\big)$	$O\big(2log(\|V\|) + log(\|E\| \cdot h) + r \cdot h\big)$
$setVProp$	$O\big(log(\|V\| \cdot P)\big)$	$O\big(2log(\|V\| \cdot P)\big)$	$O\big(2log(\|V\| \cdot P) + log(\|V\| \cdot P \cdot h)\big)$
$getVProp$	$O\big(log(\|V\| \cdot P)\big)$	$O\big(log(\|V\| \cdot P)\big)$	$O\big(log(\|V\| \cdot P) + log(\|V\| \cdot P \cdot h)\big)$

Table III
THE TIME COMPLEXITY OF THE TIMED METHODS

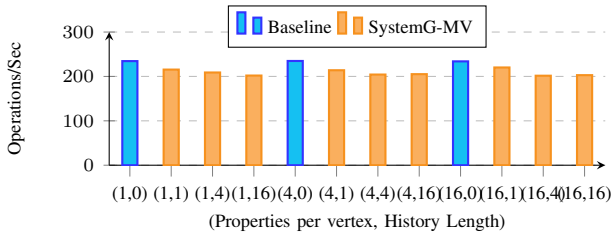

Figure 14. Performance of BFS queries on current graph.

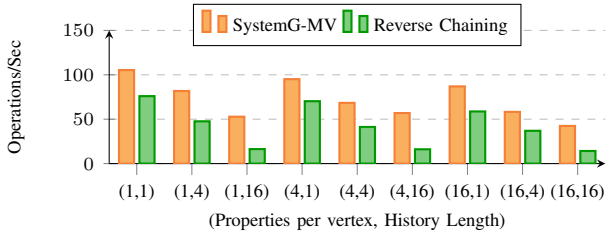

Figure 15. Performance of BFS queries on historic graph.

past history access. Even though we used the big O notation, we tried to keep the constant to highlight the differences. In the formulas, $|V|$ represents the number of vertices, $|E|$ represents the number of edges, P represents the number of properties per vertex. Also, r and h are used as multiplier variables for the number of edges between a single source-target pair, and the length of the history tables with respect to the current graph tables, respectively. For the analysis, we roughly assumed that it takes O(log(N)) time for a single lookup using a key on the backend storage, and O(1) for getting the next entry after accessing the backend storage where N is the number of items in the table. Please note that this assumption might be slightly incorrect due the implementation of the backend storage, since it does not take details into account such as caching effect.

For most of the methods, the difference between original method and the accessing the versioned graph with the current time incurs no or minimal difference. For accessing the vertices, $log(|V| \cdot h)$ is added for accessing the vertex history table of length $|V| \cdot h$.

For accessing edges between a vertex pair is similar, the cost is the sum accessing the two vertices, and also the two edge tables for incoming and outgoing edges. When reading edges, r entries between a pair of vertices have to be traversed after a lookup to the edge table, which becomes $r \cdot h$ for accessing the past version of the graph. Thus the dominating cost for reading edges would usually be $log(|E| \cdot h)$ or $r \cdot h$.

Setting properties is the only method in the table that has difference between baseline and current version access. In the baseline graph database, the property can be just overwritten. However in the versioned graph database, the property table has to be read once before writing to look for an existing entry to be migrated to the history table. When an existing entry is found, the migrating cost is $log(V \cdot P \cdot h)$ added to total for accessing the property history table.

B. Restoration Policy

In the proposed implementation of graph database, the policy of graph history access can be called a lazy restoration. All the past instance of entities are read on-demand, and open_graph(time) method simply sets the internal current_time variable. This method avoids the cost for having to materialize the whole graph, especially on small queries. However, for some large queries, restoring the full graph at the point of opening the graph (early restoration), similar to graph pool approach [11] can be more beneficial. For example, while running a PageRank on a graph with few properties for five iterations, lazy restoration scheme would require reading the history five times for each entities, while early restoration would requires only one read. In the early restoration scheme, the history of all the entities are traversed, and put into the "current" tables just like the latest data are handled in the previous explanation of the graph database. As such, queries evaluated at the specified time would have the time complexity listed in the Timed (current) column rather than the Timed (past) column of Table III.

C. Snapshot-hybrid Approach

Even though snapshots cannot provide a fine-grained time access, a hybrid approach with snapshot and history table can be used for reducing the length of the history. One possible implementation for a hybrid scheme is to partition the history entries into multiple chunks based on their timestamps and embed them into snapshots. When the queries are expected to access time points within a certain partition, the snapshot of that partition is first restored, and the history entries are read on demand. This would reduce

the h variable of the cost model in Section VIII-A, and becomes especially beneficial for accessing edges, which has a complexity proportional to h.

D. Alternative Configurations

To index history entries in the table, we generally used entity id + deletion time as their keys and creation time + the rest as their values. However, there are also other possible configurations, and we discuss the alternative choices in this section. One could choose to place the event logs, instead of keeping creation and deletion time for each entry. In such configuration, vertex history table, for example, would have the vertex id and event time as the key, and event type in the value. This is useful for a certain type of range queries, which asks for the changes that has been made over a certain range of time. However, it requires at least two reads to find out whether an entity is valid at a time point.

The radix order of the keys can be changed to optimize for more common types of queries. For example, the key for edge history can be in the order of source-time-target in order to optimize for range queries over an interval of time. Similarly, the property table's key can be ordered by v/eid-time-pid for the tracking change on a parent entity instead of a specific property instance. While putting the time before the vid for the vertex history is also possible, the trade-off is too harsh for looking for a certain vertex since potentially the whole history has to be read to find a single vertex.

The key radix problem above is caused by the fact the underlying B-tree imposes a linear ordering of the data used in the key. To provide a balanced, efficient query processing, k-d tree [6] can be used to mitigate the problem. k-d trees are known to well-support multi-dimensional range queries, and therefore a great fit for a backend storage of our graph databases. We leave this as our another future work.

IX. Conclusion

We proposed a design for time-versioned graph database, on top of an original database which uses LMDB as a back-end. Through separate current table and range-searchable timestamps, we provide almost no slowdown for accessing the up-to-date state of the graph, while experiencing graceful performance overhead for increasing the length of history. We have also shown that our implementation is superior to the naive alternative implementation. Together, they provide a scaleable implementation for the versioned graph databases. As mentioned in Section VIII, exploring richer set of alternative configurations would be our future work.

References

[1] Allegrograph. https://franz.com/agraph/support/documentation/6.2.2/.

[2] Janusgraph distributed graph database. http://www.docs.janusgraph.org/.

[3] Lightning memory-mapped database manager (LMDB). http://www.lmdb.tech/doc/.

[4] System G distributed graph database. To Be Published.

[5] Time-based versioned graphs. http://iansrobinson.com/2014/05/13/time-based-versioned-graphs/.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[7] A. Castelltort and A. Laurent. Representing history in graph-oriented nosql databases: A versioning system. In *ICDIM*, pages 228–234, 2013.

[8] C. Dyreson, F. Grandi, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, et al. A consensus glossary of temporal database concepts. *ACM Sigmod Record*, 23(1):52–64, 1994.

[9] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: a graph engine for temporal graph analysis. In *Eurosys*, page 1, 2014.

[10] A. P. Iyer, L. E. Li, T. Das, and I. Stoica. Time-evolving graph processing at scale. In *GRADES*, page 5. ACM, 2016.

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

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

[13] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.

[14] J. Lee, H. Kim, S. Yoo, K. Choi, H. P. Hofstee, G.-J. Nam, M. R. Nutter, and D. Jamsek. ExtraV: Boosting graph processing near storage with a coherent accelerator. *Proceedings of the VLDB Endowment*, 10(12), 2017.

[15] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2, 2007.

[16] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, pages 1894–1905, 2012.

[17] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient graph analytics using large multiversioned arrays. In *ICDE*, pages 363–374, 2015.

[18] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.

[19] K. H. Randall, R. Stata, R. G. Wickremesinghe, and J. L. Wiener. The link database: Fast access to graphs of the web. In *DCC*, pages 122–131, 2002.

[20] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly Media, Incorporated, 2013.

[21] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[22] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, June 1999.

[23] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *FAST*, pages 45–58, 2015.