TGraph: A Temporal Graph Data Management System

Haixing Huang Jinghe Song Xuelian Lin Shuai Ma* Jinpeng Huai
SKLSDE Lab, Beihang University, China
Beijing Advanced Innovation Center for Big Data and Brain Computing, Beijing, China
{huanghx, songjh, linxl}@act.buaa.edu.cn {mashuai, huaijp}@buaa.edu.cn

ABSTRACT

Temporal graphs are a class of graphs whose nodes and edges, together with the associated properties, continuously change over time. Recently, systems have been developed to support snapshot queries over temporal graphs. However, these systems barely support aggregate time range queries. Moreover, these systems cannot guarantee ACID transactions, an important feature for data management systems as long as concurrent processing is involved. To solve these issues, we design and develop TGraph, a temporal graph data management system, that assures the ACID transaction feature, and supports fast temporal graph queries.

1. INTRODUCTION

Graph databases have attracted attention from both academia and industry. Numerous applications, from social media analysis and traffic navigation to recommendation systems, handle a vast collection of entities with their relationships [5], which can be naturally represented by graphs. Some commercialized graph database products, such as Neo4j [6] and Titan [9], have also been developed to manage graph data and to support graph-oriented applications, which usually provide update and read operations over the latest state of graphs, and support ACID transactions, a key feature of traditional DBMS [8].

Temporal graphs (see a recent survey [2]) are a class of graphs whose nodes, edges and their properties continuously change over time. For example, traffic conditions in a road network vary from morning to evening, friendships between people in a social network evolve with time. Temporal graph databases manage temporal graph data by storing all snapshots of graphs, as user queries may focus on historical states and changing patterns of graphs.

Temporal data management started from the 80's of last century, which mainly focused on the temporal query language on temporal relational databases [7]. Temporal graph data management has recently been initiated, and several temporal graph management systems have been developed. DeltaGraph [3] stores temporal graphs using a log-based storage approach and provides snapshot queries; And G* [4] provides a distributed storage of temporal graph data. Further, how to analyze temporal graphs is also a hot research topic in database and data mining fields [10].

However, these existing temporal graph data management systems have certain limitations. First, most of them mainly support snapshot queries. For a temporal range query, DeltaGraph needs to load all the relevant data falling into the time interval from all snapshots, which makes it very inefficient. Indeed, this is typically unnecessary as many such queries only need a small part of nodes and relationships (edges). Second, these systems cannot guarantee the ACID transaction feature, which potentially leads the systems to inconsistent states when concurrent data processing is involved. Finally, we find that there is an important type of temporal graphs in real-life, whose nodes and relationships barely change while the properties of nodes and relationships change frequently. For example, roads are associated with certain properties, such as the length and the direction of the roads, and a road network topology keeps static for a long time, but the traffic condition changes frequently.

Contributions. To this end, we develop TGraph for managing temporal graphs, whose structures barely change, but associated properties change frequently. To the best of our knowledge, this is among the first temporal graph management system that is able to handle such temporal graphs and to support ACID transactions.

(1) We develop a fast and effective storage called DPS designed to maintain the frequent changing properties of nodes/relationsh in temporal graph data.

(2) We also provide a transaction manager to ensure the ACID transaction feature of temporal graph data.

(3) We finally build a prototype temporal graph data management system that supports various temporal queries.

Organization. We first describe the data model in Section 2. We then introduce the architecture and key mechanism of TGraph in Section 3. Finally, we demonstrate TGraph with an experiment study and two use cases in Section 4.

2. TEMPORAL GRAPH MODEL

A temporal graph is $G = (V, E)$, where V is a set of nodes, and E is a set of relationships. A node $v = (id, P, DynP) \in V$, where id is the identity of v in V, P is a set of static properties, and $DynP$ is a set of dynamic properties.

A static property (a concept borrowed from Neo4j) is $p = (name, value) \in P$, where $name$ is the unique name of p, and $value$ is the value of p, respectively.

CIKM'16 October 24 - November 28, 2016, Indianapolis, IN, USA

ACM ISBN 978-1-4503-4073-1/16/10.
DOI: http://dx.doi.org/10.1145/2983323.2983335

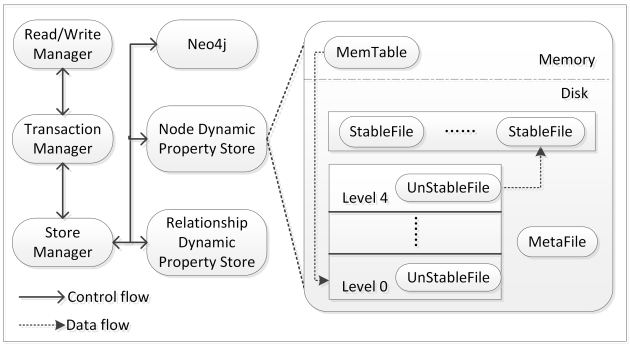


Figure 1: TGraph architecture.

A dynamic property is $dp = (name, vlist) \in DynP$, where *vlist* is a list of (t, *value*), which means that value v is valid from time t. A $vlist$=$\langle(t_1, value_1)$, $(t_2, value_2)$, $(t_3, value_3)$, ..., $(t_n, value_n)\rangle$, satisfying $t_i < t_{i+1}$ for $1 \leq i < n$. A tuple $(t_i, value_i)$ represents that the value of dp in $[t_i, t_{i+1})$ is $value_i$. For the last tuple $(t_n, value_n)$, its value of dp from t_n to *now* is $value_n$.

A relationship is $e = (id, vs, ve, ty, P, DynP) \in E$, where id is the identity of e, vs and ve are the start node and end node of e, and ty is the type of e, respectively.

3. ARCHITECTURE

TGraph is an extension of Neo4j [6], a popular, open source graph database. TGraph uses Neo4j to store nodes, relationships and static properties of nodes/relatinships, while dynamic properties of nodes/relationships is stored in another component, referred to as DPS (Dynamic Properties Storage). TGraph integrates Neo4j and DPS together to provide a complete temporal graph data management system, and its architecture is shown in Figure 1.

The Read/Write Manager detects read and write requests, and passes them to the Transaction Manager with the ACID transaction feature. The Store Manager manages the storage of nodes, relationships, static and dynamic properties, and the node and relationship dynamic properties are separately stored as two different DPS instances. We next explain the details of DPS and the transaction manager.

3.1 Dynamic Property Storage

In DPS (right side of Fig. 1), all data is first written into an in-memory data structure MemTable. It is dumped to disks when MemTable is full. DPS has three types of local files: UnStableFile, StableFile and MetaFile. Both UnStableFile and StableFile are data files to store dynamic properties. MetaFile stores the metadata of DPS, including the names of data files, and the valid time ranges of data files, which is loaded into memory when the system is started.

UnStableFiles are used to save the most recent data. When a MemTable is dumped to disks, a level 0 UnStableFile is created. Then they can be merged into a level 1 UnStableFile, and so on. Level 4 UnStableFiles can be merged into a StableFile. Each StableFile/UnStableFile covers a valid time interval. Any two valid time intervals of StableFiles/UnStableFiles have no overlaps with each other. Dynamic properties stored in a file have their time fall into the time interval of the file. The MetaFile is used to maintain the name and valid time interval of each data file.

Data Format. StableFile and UnStableFile files have the same format, and are divided into data and index segments. (1) The data segment has a number of data blocks whose sizes are set to 4KB by default, and are configurable. All data of dynamic properties is stored in data blocks. The system allocates a *proid* for a dynamic property when it is created, to identify the dynamic property within a node or a relationship. Logically, dynamic property data is stored in a list of records. A record is a tuple (*id*, *proid*, *t*, *value*), where *id* is the id of a node or a relationship, t is the valid time and *value* is the value. Each record uses (*id*, *proid*, t) as its key. As all records are stored in the ascending order of their keys, each data block is associated with a range of record keys. (2) The index segment stores the indexing information of all data blocks.

Writes and Reads on Dynamic Properties. All write operations (create, update and delete) must be logged in the MemTable first. When the MemTable is full, its data is dumped to disks, and TGraph decides whether to write a data record into a newly created UnStableFile, or to insert it into an existing StableFile or an UnStableFile, depending on the valid time of the data record.

(1) The insert operation could cost lots of IOs as records are ordered by their keys in a data file. Consider there is an insertion to a StableFile sf. To reduce the insertion IO cost, the system allocates a Buffer in memory for sf, and all subsequent insertions to sf are inserted into the Buffer. When the Buffer reaches 4MB (configurable), the system merges it with sf. When an UnStableFile is merged with another UnStableFile, their Buffers is merged as well. The system also creates a BufferFile for each Buffer as its backup in case the system crashes.

(2) For the update operation, consider a record to be updated in MemTable. The system updates it in MemTable directly. However, if it is in disks, the system transforms it into an insert operation, and adds it to MemTable. When MemTable is dumped to disks, it is added to the Buffer of the target file, and the corresponding record is finally updated during the merging process.

(3) The delete operation is similar to the update operation. The system does not delete the target record directly, but labels the record with a deletion mark and adds it to the Buffer of the target file, and the records with deletion marks in the Buffer do not participate in any merging process.

(4) All user queries first read from MemTable, then from a StableFile (or an UnStableFile) and its Buffer, and the system itself chooses and returns the right answers.

3.2 Transaction Manager

TGraph uses Neo4j's mechanism to ensure ACID guarantee with one major improvement. First, Neo4j ensures the ACID feature by the following. (1) Binding a transaction with a thread and making uncommitted data as a private member of the thread so that uncommitted data cannot be accessed by other transactions. (2) Using read-write locks on nodes/relationships to control concurrent accesses. So writes on the same data are serialized. (3) Writing all the modified data to the disk only at the commit stage of a transaction. So there will not be half-committed transaction. (4) Using the write–ahead logging technique [8] to record all operations in a transaction in case crash occurs.

Second, TGraph extends the resource lock of Neo4j to two types of resource locks: one for nodes, relationships and static properties, and the other for dynamic properties. The former is the common read-write lock that uses a node or

relationship as the unit of a lock. The latter uses a dynamic property as the unit of a lock. Unlike common read-write locks, not all read operations are blocked by the write lock on the same resource. For example, if a transaction updates a dynamic property p at time t and holds a write lock of p, and another transaction needs to read the value of p at time t', then the read transaction is not blocked if $t' < t$.

4. DEMONSTRATION

We demonstrate TGraph by comparing its performance with Neo4j and by introducing two use cases with details in the Appendix. We use a *real-life traffic dataset* recording the traffic of Beijing road network's conditions within a period of six months. The data contains nearly 110,000 roads such that joints of roads are graph nodes, and roads are relationships between nodes, which have four dynamic properties (travel time, vehicle count, traffic condition, jammed segment count) whose values are updated every 5 minutes and some static properties such as length and direction.

All experiments were conducted on a machine with 2 Intel Xeon E5-2630 2.4GHz CPUs and 64 GB of Memory, running a 64 bit Windows 7 Ultimate system.

(1) Performance Evaluation. We compare the performance between TGraph and Neo4j to show the advantages of TGraph's performance in Figure 2. The reasons why we chose Neo4j are its popularity and support of ACID transactions, while other temporal graph management systems rarely do. In Neo4j, we encode a dynamic property into an integer array whose size is around 5MB, and store it as a static property of a relationship since Neo4j supports the array as the value type of a property.

Figure 2(a) shows the result of write operations. TGraph makes 52,551 write operations per second on average, which is 20 times faster than Neo4j (2,621/s). The write speed of TGraph does not slow down as the write operation continues while Neo4j decreases significantly.

Figures 2(b) and 2(c) show the results of time point and time range queries. TGraph can finish more read operations than Neo4j (12,500/s vs. 439/s and 10,858/s vs. 403/s for time point and time range queries, respectively). That is because Neo4j needs to load the entire array from disks for any query covering this property. Note that the length of an array becomes very large when the value of the dynamic property changes frequently. However TGraph stores dynamic properties in DPS which is designed for storing frequent changing properties.

Figure 2(d) shows the result of concurrent capacities. We use a different number of threads to read or write randomly from/to both systems. Every transaction only contains a single operation, either read or write. We let each transaction sleep for 10ms to replace the cost of system IO to see the improvement of the transaction manager in TGraph without the disturbance of the differences between the storage methods of the two systems. The concurrent capacity of TGraph transaction manager is 1.13 times better than the one of Neo4j, mainly because the resources lock of TGraph is optimized for read/write operations of temporal graphs.

(2) Use Cases. We further demonstrate TGraph by building a traffic data management system on top of TGraph, by making use of TGraph APIs. The system supports historical traffic condition queries, historical traffic condition aggregate queries, and temporal shortest path queries. All these capacities are supported by TGraph's time point and time range query abilities. The interface of TGraph is shown in the Appendix, with an example on how to use its APIs.

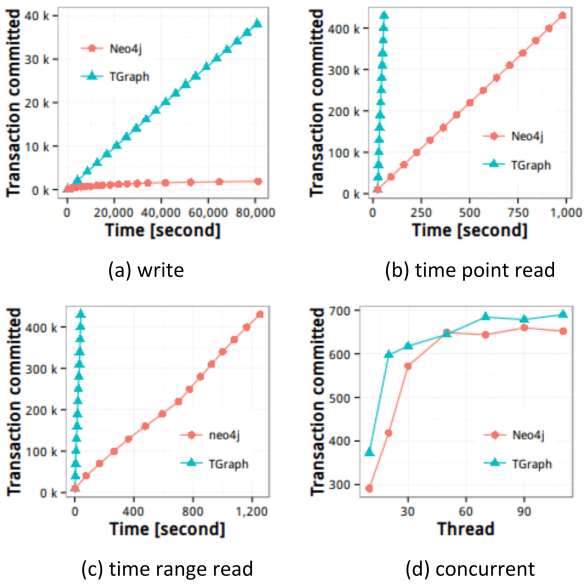


Figure 2: Performance comparison: TGraph vs. Neo4j.

5. CONCLUSIONS

We have introduced TGraph for managing temporal graphs whose structures barely change, but the associated properties change frequently. TGraph has been equipped with DPS, a fast and effective storage to maintain the dynamic properties of temporal graph data, and a transaction manager to ensure the ACID transaction feature for temporal graph data. Experiments on real traffic data have shown that TGraph is more suitable to manage temporal graph data than Neo4j, and, moreover, we have also illustrated use cases on how to utilize TGraph to develop applications.

Acknowledgment This work is supported in part by 973 program (2014CB340300), NSFC (61322207&61421003), Special Funds of Beijing Municipal Science & Technology Commission, and MSRA Collaborative Research Program. For any correspondence, please refer to Shuai Ma.

6. REFERENCES

[1] S. E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412, 1969.
[2] P. Holme and J. Saramäki. Temporal networks. *Physics reports*, 519(3):97–125, 2012.
[3] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, 2013.
[4] A. G. Labouseur, P. W. Olsen, and J.-H. Hwang. Scalable and robust management of dynamic graph data. In *BD3@VLDB*, 2013.
[5] S. Ma, J. Li, C. Hu, X. Lin, and J. Huai. Big graph search: challenges and techniques. *FCS*, 10(3):387–397, 2016.
[6] Neo4j. http://neo4j.com/.
[7] G. Özsoyoğlu and R. T. Snodgrass. Temporal and real-time databases: A survey. *TKDE*, 7(4):513–532, 1995.
[8] R. Ramakrishnan and J. Gehrke. *Database management systems*. Osborne/McGraw-Hill, 2000.
[9] Titan. http://thinkaurelius.github.io/titan/.
[10] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *PVLDB*, 7(9):721–732, 2014.

APPENDIX

A. TRAFFIC DATA MANAGEMENT SYSTEM

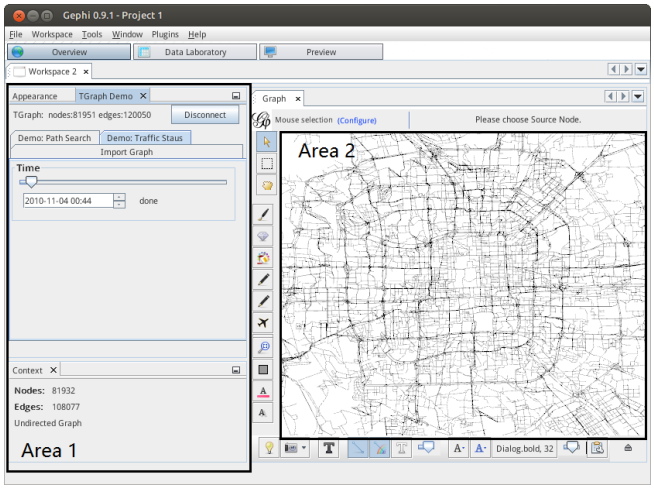


Figure 3: User interface of a traffic data management system

An Example Program of TGraph
1. GraphService service = new GraphService(...);
2. Transaction tx = service.beginTx();
3. Node tom = service.getNodes("name", "tom").get(0);
4. tom.setDynProperty("work place", "2016/1/01", "Google");
5. tom.setDynProperty("salary", "2016/1/01", 20000);
6. tx.sucess();
7. tx.close();
8. ···
9. tx = service.beginTx();
10. tom.getDynProperty("work place", "2016/1/01");
11. tom.getDynPropertyMax("salary", "2016/1/01", "2016/6/01");
12. tx.success();
13. tx.close().

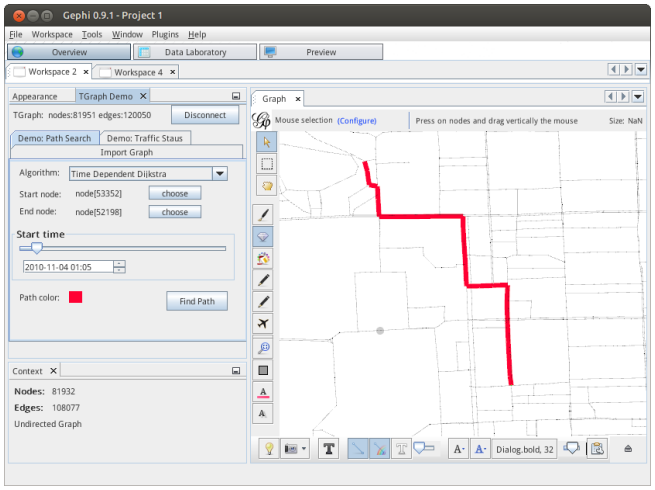
Table 1: An Example Program of TGraph

Figure 3 is the user interface of traffic data management system built on top of TGraph, using Gephi (https://gephi.org). It has two major areas. (1) Area 1 is the control panel from where users choose the type of queries, and set the conditions of queries, and (2) road network topologies and query answers are shown at area 2, which supports zoomIn and zoomOut commands to present different levels of network details.

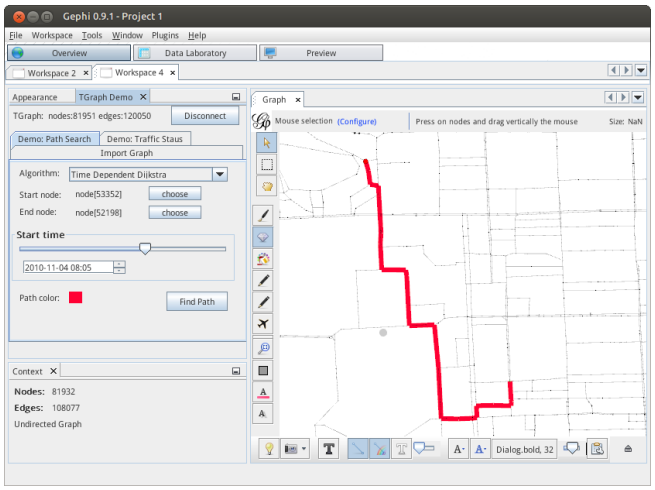
If a user wants to query the traffic situation at a history time point, the user needs to set the time point at area 1 and to trigger the query. The system then sends the query to TGraph and shows the corresponding results returned from TGraph on the road network.

If a user wants to find a temporal shortest path between two nodes, the user needs to select a start node and an end node by clicking them at area 2, and sets a departure time at area 1. The system runs a time-dependent *generalized Dijkstra* algorithm [1] to compute a path with the earliest arrival time, and highlights the resulting path at area 2.

Figure 4 is an example that tells the temporal shortest pathes between two nodes at different departure time. Figure 4(a) illustrates the path, departed at 1:05 am, recommended by TGraph, which is indeed the same as the shortest



(a) Temporal shortest path departing at 1:05 am



(b) Temporal shortest path departing at 8:05 am

Figure 4: Temporal shortest path queries: given the same two nodes with the different departure time, return two different pathes.

distance path since all traffic conditions are smooth during that time; Figure 4(b) illustrates the recommended path departed at 8:05 am in the traffic peak time, and the system returns a different path that avoids jammed roads.

B. PROGRAMMING WITH TGRAPH APIS

TGraph provides a full package of operations to write and query both static and dynamic graph data, and to assure ACID transactions.

An example program is provided in Table 1. The program first initializes the TGraph service (line 1), and starts a transaction to find the first node whose "name" property is "tom" (lines 2, 3). Then it sets the property "work place" of the node "tom" to "Google" on date "2016/1/01", and the salary of "tom" to "20000" on date "2016/1/01" (lines 4, 5). After that, it closes the transaction (lines 6, 7). (3) Finally, it starts a new transaction to query information from those modifications (line 9). It finds the "work place" of "tom" on date "2016/1/01" (line 10), and the max salary of "tom" from dates "2016/1/01" to "2016/6/01" (line 11).