A Graph Database Storage Engine for Provenance Graphs

Changhong Liu

School of Computer Science and Engineering,
University of Electronic Science and Technology,
Chengdu, China 610000
Email: 314979677@qq.com

Hancong Duan

School of Computer Science and Engineering,
University of Electronic Science and Technology,
Chengdu, China 610000
Email: duanhancong@uestc.edu.cn

Abstract—**The rapid development of high-speed networks has created a massive amount of data. Storing and mining such data is of great research value. Knowledge graphs and graph databases have widely been studied and applied as an effective means to mine the associated data in the past few years. Provenance graphs provide powerful ways to observe the changes in a graph, especially in graph analysis. The update operation will produce massive provenance graphs from a given graph as time goes on. It is a challenge to store and query these massive provenance graphs efficiently. Meanwhile, the query performance itself must be guaranteed. To address this challenge, this paper presents a graph database storage engine called T-GDB (Temporal dimension - Graph Database). This system binds the topology of the graph to each vertex in the graph and rebuilds the graph in real-time when analyzing the graph. T-GDB can analyze the changes in a graph over time and can also access the provenance of the specified graph through the index tree. T-GDB can support these application scenarios such as the knowledge reasoning of knowledge graphs and the information mining for specified graphs. This paper describes the format of data storage, the index, and the implementation of this system. Finally, this paper compares the proposed graph database storage engine to several existing mainstream graph databases to verify the feasibility and efficiency of this design. Our experimental results demonstrate that the proposed graph database storage engine has better performance and more efficient graph analysis than existing methods.**

Keywords–Graph Database; Graph Analytics and Storage; Provenance Graphs.

I. INTRODUCTION

In the age of big data, big graph analysis has widely been studied in recent years because of its many applications in a wide variety of practical fields. Many algorithms of graph computing are NP-hard (non-deterministic polynomial-time hard) problems such as Graph Partition [1]. It is challenging to study how to store graph-structured data and reduce the computing latency for graph computing. As a research field of artificial intelligence, knowledge graphs [2] play an important role in intelligent data analysis. Knowledge graphs can be stored in Resource Description Framework (RDF) [3], XML (Extensible Markup Language) [4], or other formats. Property graphs [5] (see Figure 2) are also effective data models for applying knowledge mining in graph databases. Graph databases can efficiently query the properties attached to vertices and edges of the graph, while RDF is less effective at doing that. NoSQL (Not Only Structured Query Language) databases have several storage types: Key-Value like Redis Graph [6], Document like CouchDB [7], Column-oriented like Bigtable [8] and graph database like Neo4j [9]. Therefore, graph database is one type of NoSQL databases. However, existing graph databases still employ several storage formats. In this paper, the storage format of the storage engine is similar to Key-Value. The simple statement queries of graph databases do not care much about the memory usage. However, memory usage is vital for graph analysis because it always traverses the whole graph. To address this, Trinity [10] presented an optimized memory management for graph-structured models. Although graph databases have great advantages in dealing with relationships between data, graph indexing [11] is also necessary to speed up graph computing. This paper's contributions are as follows:

- Propose a unique tree-structured index for provenance graphs and an efficient graph storage model for graph traversal.
- Provide a storage engine architectural design. This system can read, write and analyze the graph-structured data conveniently and quickly.

The rest of this paper is structured as follows. This paper describes the background of the system and the related work in Section II. In Section III, this paper details the storage format of the system, both in memory and on the disk. In Section IV, this paper provides the architecture and implementation of the system. This paper discusses the performance of T-GDB and compares the results with other graph databases in Section V. This paper discusses the future work related to the research in Section VI.

II. BACKGROUND AND RELATED WORK

In many existing graph databases, the changes in a graph over time can not be queried. Graph databases usually deal with the relationships between data. However, many existing graph databases can not do anything about the relevant causality of data. For example, the process of knowledge reasoning will produce the relevant causality of data. Data provenance has widely been studied in the field of databases. Provenance graphs provide powerful ways to analyze the graph-structured data like Ariadne [12]. However, developers did not specially design effective storage for provenance graphs in graph databases. Knowledge mining must be considered in our system. Knowledge mining can mine the potential information of the graph-structured data and can also deduce new knowledge through knowledge mining algorithms. Cook et al. [13] present the details of many kinds of graph mining algorithms. In the research field of knowledge graphs, DBpedia [14] is the leader in knowledge storage. Freebase [15] is a graph database for building human knowledge. However, these previous studies can not query provenance graphs. Many existing graph databases can also not query knowledge graphs or lack support for knowledge mining. There are some major graph databases, such as Neo4j, TigerGraph [16] and JanusGraph [17]. These major graph databases are not suitable for storing knowledge graphs or provenance graphs. Neo4j uses an orthogonal list to represent the graph-structured data. JanusGraph uses an adjacency matrix to represent the graph-structured data. However, T-GDB uses an array to represent the graph-structured data. Neo4j uses the ID of vertices or edges as the graph indexing. Neo4j reads the graph-structured data from disk through the graph indexing. JanusGraph uses the Key-Value to read the graph-structured data from disk. These existing graph databases can not compactly store the graph-structured data according to the characteristics of graph. The compact storage can help graph databases read the graph-structured data from disk sequentially. T-GDB provides a special design for compact storage. The final goal of T-GDB is to meet both OLAP (On-Line Analytical Processing) and OLTP (On-Line Transaction Processing) requirements.

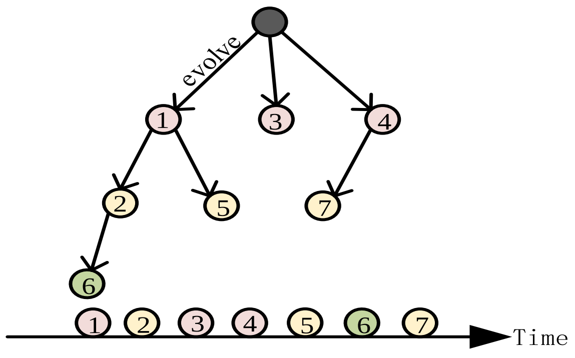


Figure 1. The logical relationship of provenance graphs.

Knowledge reasoning [18] is a key technology in knowledge graphs. Knowledge graphs deduce new knowledge over a given graph according to different rules. Knowledge reasoning may need to access the provenance of the specified graph and reason repeatedly. That (see Figure 1) is a good explanation for provenance graphs. Pugliese et al. [18] proposed a temporal RDF model. Lu et al. [19] proposed a temporal data storage based on TDSQL. Time is a key metadata to query the changes in data according to [18] and [19]. Leskovec et al. [20] describe the changes in graph over time. Knowledge reasoning can form the logical relationship of provenance graphs (shown in Figure 1) according to the above studies. Each node of the tree-structure represents an index file for the special graph. Time properties are also important for the graph databases to observe the subtle changes in a graph.

III. Data Model

The storage engine uses property graphs as the data model in this paper. Property graphs are directed graphs consisting of vertices, edges, and properties (see Figure 2). Labels and relationship types are particular properties of vertices and edges, respectively. Graph queries can filter out a lot of useless data according to labels and relationship types. The time is also a unique property in our storage engine. Because these particular properties always play an essential role in graph query, our storage engine stores them in different formats. This storage engine has mainly two parts: memory storage format and disk storage format. This paper will describe them in detail below.

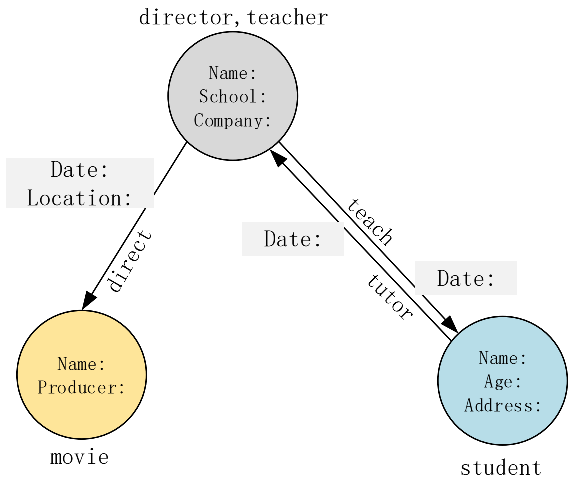


Figure 2. An example of property graphs.

A. Memory Storage Format

Many existing graph databases usually use adjacency lists or cross lists to store the graph-structured data. However, we use arrays to store the graph-structured data in this paper. Every vertex and edge of property graphs has a fixed-length byte in arrays. Each vertex accesses its neighbors through the array address. The time complexity is O(1) when this system traverses property graphs. The storage format is an excellent benefit for graph queries. Because all delete, update, and insert operations of our graph database are done in an append manner, this system has no restrictions on storing graph-structured data in arrays. The graph has three parts: vertices, edges, and topologies. The graph-structured data will be serialized from disk to memory when graph queries need to access the specified graph. However, simple graph queries can get data directly through graph indexing without rebuilding the graph.

TABLE I. The structure of the vertex.

type:	uint32	uint32	uint32	uint32
vertex array:	Pid+VertexId	TopoOffset	Flag+OEOffset	Time

The structure of the vertex is shown in Table I. All vertices of a graph are stored in a vertex array. Each vertex has a fixed-length byte in the vertex array. The Pid is short for partition id. Because a big graph will be divided into many subgraphs, the partition id is the id of one subgraph. The VertexId is a unique vertex number in one subgraph. The TopoOffset (Topology Offset) is the topology index of topology array. The Flag field is reserved for particular purposes. The OEOffset (OutEdge Offset) is an offset relative to TopoOffset. This system needs to distinguish incoming edges and outcoming edges by the OEOffset. The Time is a property of the vertex. This system can observe the subtle changes in a graph through the Time.

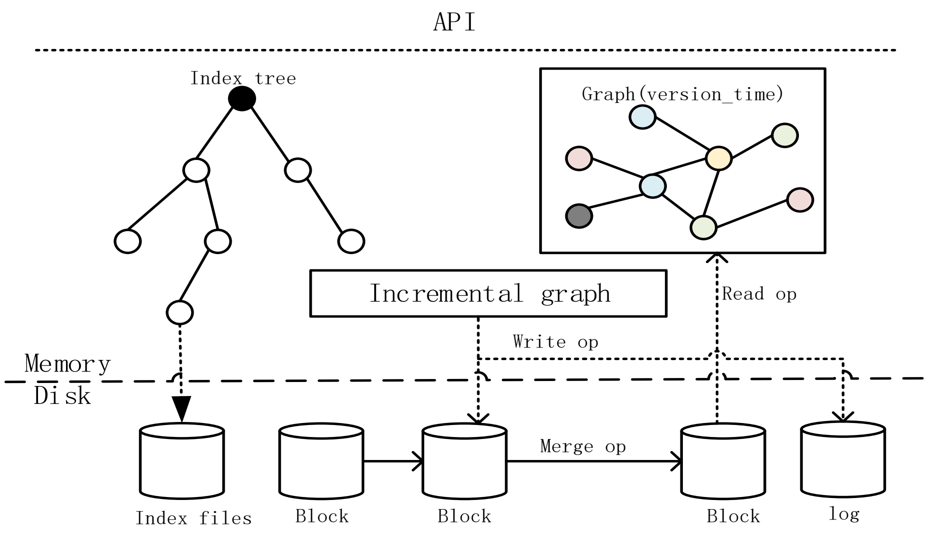


Figure 3. The framework of T-GDB.

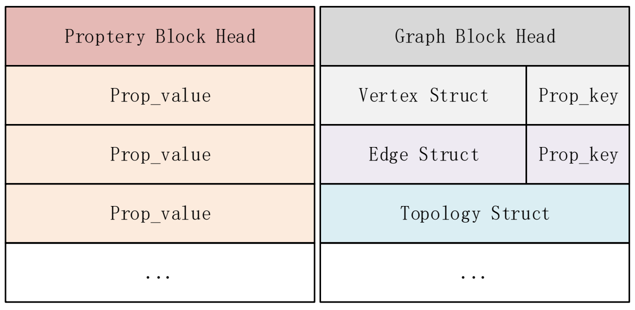
TABLE II. THE STRUCTURE OF THE EDGE.

type:	uint32	uint32	uint32	uint32	uint32
edge array:	Pid+SrcId	Flag+DistId	EdgeId	RelType	Time

The structure of the edge is shown in Table II. All edges of a graph are stored in an edge array. Each edge has a fixed-length byte in the edge array. The SrcId and DistId are the id of source and destination of the edge, respectively. They are the vertex index of vertex array. This system will assign a unique EdgeId number to each edge. The EdgeId is the edge index of edge array. Relationship types and labels will be stored in a dictionary mode. The RelType is the dictionary number in edge array.

TABLE III. THE STRUCTURE OF THE TOPOLOGY.

type:	uint32	uint32	uint32	uint32	uint32
topo array:	VertexId	Flag	Label	OutEdgeId	InEdgeId



Proptery Block Head
Prop_value
Prop_value
Prop_value
...

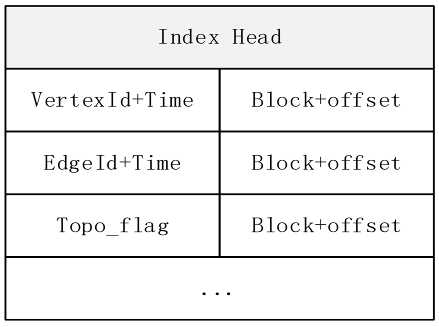
Graph Block Head	
Vertex Struct	Prop_key
Edge Struct	Prop_key
Topology Struct	
...	

Figure 4. Left is the storage of the graph, and Right is the properties of the graph.

The structure of the topology is shown in Table III. The topo array is an array containing all topologies of a graph. The Label is the dictionary number of labels. The OutEdgeId is the id of outcoming edges. The InEdgeId is the id of incoming edges. The topology is attached to the vertex. The state of topologies will be changed depending on the state of vertices and edges. Meanwhile, this system can finish graph queries within a limited time through the topology structure.

B. Disk Storage Format

The graph-structured data is stored in 4G-sized file blocks (see Figure 4). The size of file blocks is 4G because of the uint32 type. Meanwhile, our system will merge data on disk periodically to speed up graph queries. It is also beneficial to implement multiple replicas with 4G-sized file blocks. The latest file block holds the newest data because new data is appended to the existing one. The period information of file



Index Head	
VertexId+Time	Block+offset
EdgeId+Time	Block+offset
Topo_flag	Block+offset
...	

Figure 5. The index files.

blocks can help this system speed up searching provenance graphs. This system can directly access the properties of vertices and edges through the prop_key.

C. Index File

In this system, there is a unique index for provenance graphs. There are parallel meanings and chronological order between provenance graphs, according to the Figure 1. Our storage engine stores the relationship between provenance graphs in a multi-fork tree. Each node of the multi-fork tree is an index file. An index file may be full or incremental index for a graph. The head of index file has some important basic informations. This system can directly access the graph-structured data through index files (see Figure 5). Index files play a crucial role in building the graph or accessing partial graph. Index files are compressed to reduce storage overhead according to the contents of index files.

IV. Implementation

In this section, this paper details our storage engine implementation written in C++ (see Figure 3). The core design of our system reflects the features of provenance graphs. So far, we have only implemented a stand-alone system. In future work, we will implement a distributed graph database with the ability to handle large-scale storage and graph computing. The architecture of our storage engine is straightforward. The architecture has three major parts: reading, writing and merging the graph-structured data. The components of this system are described in detail below.

1) Index Tree: The index tree is a critical component in our storage engine. Each node of the index tree includes an index file and the basic informations of provenance graphs. The path from the root node to the leaf node in the index tree represents the changes of provenance graphs. There are two storage forms for index files. One is the incremental index based on the parent index file. The other is the full index for a graph. The form of index files depends on the changes of provenance graphs. It takes a little time to read index files because of the serialization of index files. The index tree is beneficial for this system to observe the changes of provenance graphs base on the timeline.

2) Updating and Building Graph: Although the graph-structured data is updated by appending data, there is still a memory buffer for a graph named Incremental graph. The Incremental graph sorts the graph-structured data according to the time in memory and puts the graph-structured data on disk. The Incremental graph can store the same provenance graphs together. It can reduce the reading time by reading data in micro blocks. Meanwhile, the update operation must be logged to ensure that the data can be recovered in the event of a system crash.

Simple statement queries typically access a part of the graph. This system can finish simple statement queries by the index of vertices or edges. This system reads the particular provenance graphs in micro blocks according to index files. The size of micro blocks depends on the distribution information of graph-structured data on disk. This system can batch load the graph when it needs the whole graph. Our storage engine is also very efficient for graph computing.

3) Merge Block: There will be hot and cold data because of the graph changes based on the timeline. The fragmented data of a graph is distributed across many file blocks. Therefore, this system will regularly merge the data of a particular graph on disk. The merge operation does not affect the previous provenance graphs. At the same time, this system removes the unused graph-structured data to increase disk utilization. The merge operation is very effective for reading data.

V. Performance Evaluation

In the section, this paper presents experiments to demonstrate the performance of the proposed system. These experiments were based on a machine with Intel(R) Xeon(R) CPU e5-2603@1.80GHz, ubuntu 16.04.10 server, 96GB RAM, and 300G SSD (DELL PERC H310 2.12). Because this system is only a stand-alone version now in this paper, all of the following experiments were tested on a single-core CPU to achieve fairness.

The datasets having Graph500 [21] and com-Orkut [22] for experiments are from the website of public datasets (see Table IV). This paper performed the experiments according to the benchmark of TigerGraph [23]. Neo4j, TigerGraph, and JanusGraph were compared in the following experiments. The version of Neo4j is community-3.4.17. The version of TigerGraph is 2.5.0-developer. The version of JanusGraph is 0.2.1-hadoop2. This paper evaluates the performance of each system according to three query types:

TABLE IV. The information of datasets

data:	*vertices*	*edges*	*Description*
Graph500	2396019	67108864	Synthetic Kronecker Graph
com-Orkut	3072441	117185083	Orkut online social network

A. The common query in graph database

The most common queries are the one-hop traversal of the graph in graph databases. It means that the one-hop traversal operation is executed from a source vertex to destination vertex through the edge. Then queries can access the properties of vertices or edges during one-hop traversal. The other common query is the three-hop traversal of the graph. However, it puts more pressure on the system.

This paper made ten thousand initial vertices to Graph500 and five hundred thousand initial vertices to com-Orkut in one-hop traversal, respectively. Figure 6(a) and Figure 6(b) are the result of a one-hop traversal query for Graph500 and com-Orkut, respectively. Because the three-hop traversal can almost traverse the whole graph, this experiment only made ten initial vertices to Graph500 and com-Orkut in three-hop traversal. Figure 7(a) and Figure 7(b) are the result of a three-hop traversal query for Graph500 and com-Orkut, respectively. It can seen that our system has an absolute advantage in the one-hop query of Graph500 and com-Orkut from Figure 6. Because our system does not have a cache yet, a vertex or edge will be reread from disk each time. TigerGraph has a built-in memory component that benefits from its data compression technology to reduce the overhead of disk. Therefore, our system is a little bit slower than TigerGraph in the three-hop traversal query. However, our system still has more advantages than the comparative databases.

B. The graph analysis

There are many complex queries, such as PageRank, SSSP (Single Source Shortest Path), WCC (Weighted Community Cluster). This experiment chose the classic PageRank [24] algorithm. Figure 8 and Figure 9 are the results of the PageRank query for Graph500 and com-Orkut, respectively. It can see from Figure 8 and Figure 9 that our system still has a great advantage in graph computing.

C. The query of provenance graphs

Our system has better performance in graph queries and graph analysis from the above experimental results. Different knowledge graphs can be deduced according to different rules

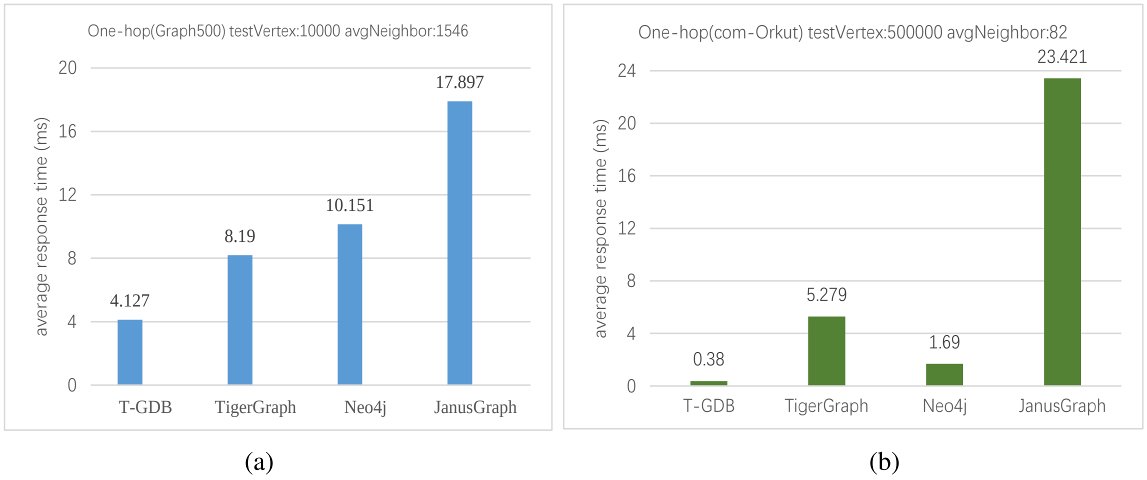


Figure 6. (a): The average response time of one-hop query for Graph500. (b): The average response time of one-hop query for com-Orkut.

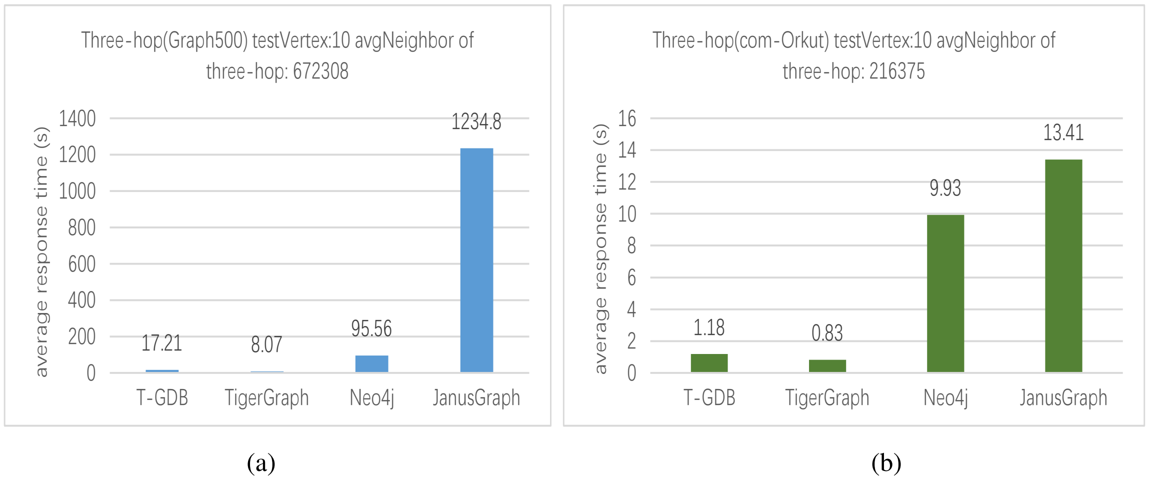


Figure 7. (a): The average response time of three-hop query for Graph500. (b): The average response time of three-hop query for com-Orkut.

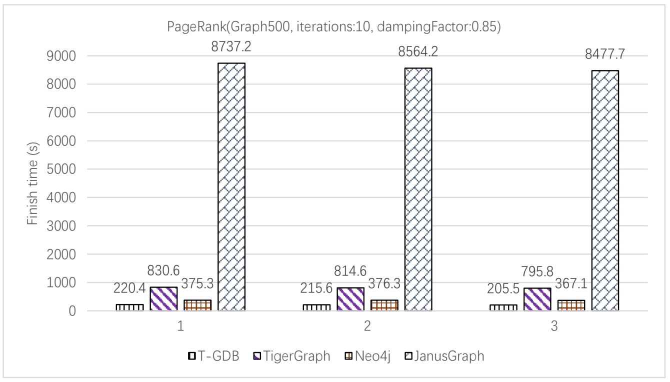


Figure 8. The finish time of PageRank having 10 iterations in graph500, Test three times.

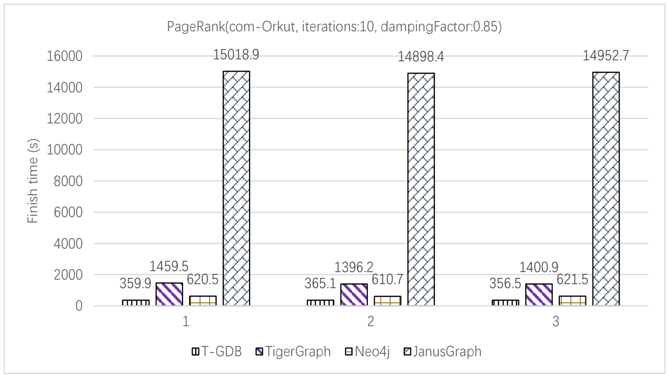


Figure 9. The finish time of PageRank having 10 iterations in com-orkut, Test three times.

in knowledge reasoning. The subtle changes of the graph can be observed through time properties. Because other graph databases do not support this kind of queries, the paper only does this queries experiment on our system. The experiment reads different provenance graphs from massive provenance graphs. For example, the Gaph500 produces many provenance graphs over a period of time (see Figure 1). This system randomly updates the time properties of vertices or edges without changing the size of the Graph500.

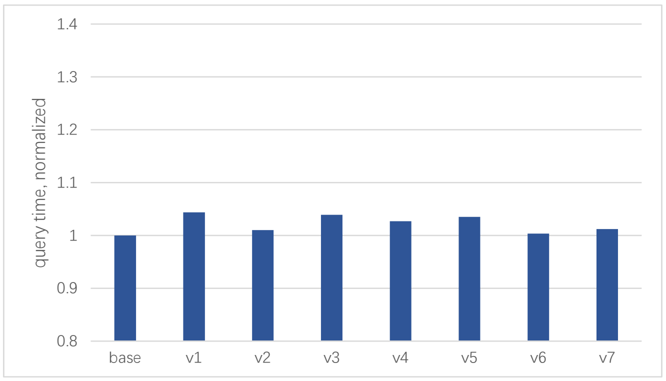


Figure 10. The cost of getting the different provenance graph.

The result of the query is shown in Figure 10. The costs are almost the same when this system reads different provenance graphs. The performance of our system does not change with the amount of data and the length of time, only with the size of the graph. Meanwhile, this system can detect partial changes of the graph through the time properties. This experiment demonstrates that the storage engine is useful for storing provenance graphs in this paper.

VI. Conclusion and Future Work

In this work, this paper proposed T-GDB, a high-performance graph database storage engine for provenance graphs. This system has a unique design to store provenance graphs efficiently without affecting the performance of graph queries and graph computing. We presented the index tree to apply the function of the provenance graphs. In this system, both index and data are stored in an append mode. The append mode is effective to observe the changes in a graph over time. Meanwhile, time plays a critical role to observe subtle changes in a graph. Although our system does not fully support the applying function of the time-series databases, it is a key to support the graph query that having time properties. Another critical point is that our system can support writing effectively because of updating the data in an appended mode.

In future work, we will focus on implementing a distributed graph database. We will ensure the data fault tolerance and the consistency of the distributed graph database. Meanwhile, we will support the application needs of artificial intelligence as much as possible.

Acknowledgment

Firstly, the authors would like to express their gratitude for the support from colleagues. Finally, they would also like to thank the benchmark of the TigerGraph.

References

[1] T. Ayall, H. Duan, and C. Liu, “Edge property based stream order reduce the performance of stream edge graph partition,” Journal of Physics: Conference Series, vol. 1395, 2019, p. 012010.

[2] L. Ehrlinger and W. Wolfram, “Towards a definition of knowledge graphs,” in Joint Proceedings of the Posters and Demos Track of 12th International Conference on Semantic Systems, 2016.

[3] F. Manola, E. Miller, and B. McBride, “Rdf primer,” W3C recommendation, vol. 10, no. 1-107, 2004, p. 6.

[4] C. C. Aggarwal, N. Ta, J. Wang, J. Feng, and M. Zaki, “Xproj: a framework for projected structural clustering of xml documents,” in Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2007, pp. 46–55.

[5] M. A. Rodriguez and P. Neubauer, “Constructions from dots and lines,” Bulletin of the American Society for Information Science and Technology, vol. 36, no. 6, 2010, pp. 35–41.

[6] “Redis Graph.” URL: https://github.com/tblobaum/redis-graph/ [accessed: 2020-02-08].

[7] J. Mondal and A. Deshpande, “Managing large dynamic graphs efficiently,” in Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, 2012, pp. 145–156.

[8] F. Chang et al., “Bigtable: A distributed storage system for structured data,” Acm Transactions on Computer Systems, vol. 26, no. 2, pp. p.1–26.

[9] I. Robinson, J. Webber, and E. Eifrem, Graph databases. ” O’Reilly Media, Inc.”, 2013.

[10] B. Shao, H. Wang, and Y. Li, “Trinity: A distributed graph engine on a memory cloud,” in Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM, 2013, pp. 505–516.

[11] X. Yan, P. S. Yu, and J. Han, “Graph indexing: a frequent structure-based approach,” in Proceedings of the 2004 ACM SIGMOD international conference on Management of data. ACM, 2004, pp. 335–346.

[12] V. Papavasileiou, K. Yocum, and A. Deutsch, “Ariadne: Online provenance for big graph analytics,” in Proceedings of the 2019 International Conference on Management of Data, 2019, pp. 521–536.

[13] D. J. Cook and L. B. Holder, Mining graph data. John Wiley & Sons, 2006.

[14] S. Auer et al., “Dbpedia: A nucleus for a web of open data.” in Semantic Web, International Semantic Web Conference, Asian Semantic Web Conference, Iswc + Aswc, Busan, Korea, November, 2007.

[15] K. Bollacker, P. Tufts, T. Pierce, and R. Cook, “A platform for scalable, collaborative, structured information integration,” in Intl. Workshop on Information Integration on the Web (IIWeb07), 2007, pp. 22–27.

[16] A. Deutsch, Y. Xu, M. Wu, and V. Lee, “Tigergraph: A native mpp graph database,” arXiv preprint arXiv:1901.08248, 2019.

[17] “Compose for JanusGraph,” URL: https://www.ibm.com/cloud/compose/janusgraph/ [accessed: 2020-02-08].

[18] A. Pugliese, O. Udrea, and V. Subrahmanian, “Scaling rdf with time,” in Proceedings of the 17th international conference on World Wide Web. ACM, 2008, pp. 605–614.

[19] W. Lu et al., “A lightweight and efficient temporal database management system in tdsql,” Proceedings of the VLDB Endowment, vol. 12, no. 12, 2019, pp. 2035–2046.

[20] J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graphs over time: densification laws, shrinking diameters and possible explanations,” in Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining. ACM, 2005, pp. 177–187.

[21] “Graph500 Large Network Dataset Collection.” URL: https://graph500.org/ [accessed: 2020-02-08].

[22] “Stanford Large Network Dataset Collection.” URL: http://snap.stanford.edu/data/ [accessed: 2020-02-08].

[23] “Benchmark for TigerGraph.” 2018, URL: https://www.tigergraph.com/benchmark/ [accessed: 2020-02-08].

[24] L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: Bringing order to the web.” Stanford InfoLab, Tech. Rep., 1999.