A Key-Value Based Approach to Scalable Graph Database *

Zihao Zhao[1,2], Chuan Hu[1,2], Zhihong Shen[1(✉)], Along Mao[1,2], and Hao Ren[1]

[1] Computer Network Information Center, Chinese Academy of Sciences, China
[2] University of Chinese Academy of Sciences, China
{zhaozihao,huchuan,bluejoe,almao,rh}@cnic.cn

Abstract. An increasing number of applications are modeling data as property graphs. In various scenarios, the scale of data can differ significantly, ranging from thousands of nodes/relationships to tens of billions of nodes/relationships. While distributed native graph databases can cater to the management and query requirements of large-scale graph data sets, they tend to be relatively cumbersome for small-scale data sets. This motivates us to develop a lightweight, scalable graph database capable of handling data across different scales. In this paper, we propose a method for constructing a graph database based on key-value storage, outlining the process of mapping graph data to key-value storage and executing graph queries on the key-value storage. We implemented and opensourced a graph database based on RocksDB, namely KVGDB, which can manage data in an embedded fashion and be easily scaled to distributed environments. Experimental results demonstrate that KVGDB can effectively meet the management and query requirements of graph data sets, even at the scale of billions of nodes/relationships.

Keywords: Graph Database · Graph data · Key-Value Database.

1 Introduction

Graph databases have emerged as a powerful tool for modeling and analyzing complex relationships between data entities in various applications, such as social networks [6] and knowledge graphs [2]. The data scale in these applications varies greatly, ranging from thousands of nodes/relationships to tens of billions of nodes/relationships. Traditional distributed native graph databases, such as TigerGraphDB [3] and ByteGraph, are based on distributed environments and can be cumbersome when dealing with small-scale datasets. Cloud databases like Amazon NeptuneDB [1] cannot be deployed locally and are difficult to meet the requirements of embedded applications. We are motivated to develop a lightweight, scalable graph database capable of handling data across different scales. Key-value databases store data in the form of key-value pairs,

* This work was supported by the National Key R&D Program of China(Grant No.2021YFF0704200) and Informatization Plan of Chinese Academy of Sciences(Grant No.CAS-WX2022GC-02)

boasting excellent scalability. They can be used in embedded environments as well as expanded to distributed environments to address storage and query requirements of large-scale datasets. Additionally, key-value databases exhibit high performance, with the cost time of prefix search unaffected by the scale of data.

This paper proposes an key-value based approach for building scalable graph databases that can efficiently manage and query graph data of various scales.

2 Methodology

In key-value databases, both the key and the value are byte arrays. Taking RocksDB as an example [4], it manages data based on Log-Structured Merge Trees (LSM). Each write operation generates a memtable in memory, which, upon reaching a certain size, is written to an SST file on disk. By default, the key-value pairs in the SST file are sorted by their keys. This sorting scheme enables key-value databases to achieve good performance in precise search and prefix search operations. Typically, a prefix iterator is used for prefix search with a time complexity of O(m+k), where m is the number of keys satisfying the prefix condition, and k is the length of the longest key. Therefore, when designing storage formats and retrieval methods for graph data on key-value databases, it is crucial to fully exploit the inherent data order and the fast prefix search capabilities of key-value databases.

2.1 Storage

Suppose a property graph could be simply represented as $G =< N, R >$, where N is the set of nodes and R is the set of relationships (a.k.a. edges). The key-value storage model (as illustrated in Fig. 1) of G could be represented as $KVG =< NS, NLS, RS, RTS, ORI, IRI, PID, PI >$, where:

- NS: NodeStore, where the key is the combination of LabelID and NodeID, and the value is a byte array containing all the property information of the node. If a node has m labels (m >1), then in the NodeStore, the node is stored as m key-value pairs, each corresponding to a label. Specifically, if a node has no label, the storage engine will set its LabelID to an ID representing an empty label.
- NLS: NodeLabelStore, it stores the label information of nodes, where the key is the combination of NodeID and LabelID, and the value is blank.
- RS: It is the RelationshipStore, where the key is the RelationshipID (i.e. RelID in Fig. 1), and the value is a byte array containing all the property information of that relationship.
- RTS: RelationTypeStore, where the key is the combination of TypeID and RelationshipID of a relationship, and the value is an empty byte array.
- ORI: OutRelationIndex, it is the outgoing edge index, built for relationships to accelerate graph query processing for a specific relationship direction. The key consists of source node ID (SrcID), relationship type ID (TypeID) and destination node ID (DstID) in order, and the value is the relationship IDs for all the relationships correspond to the key.

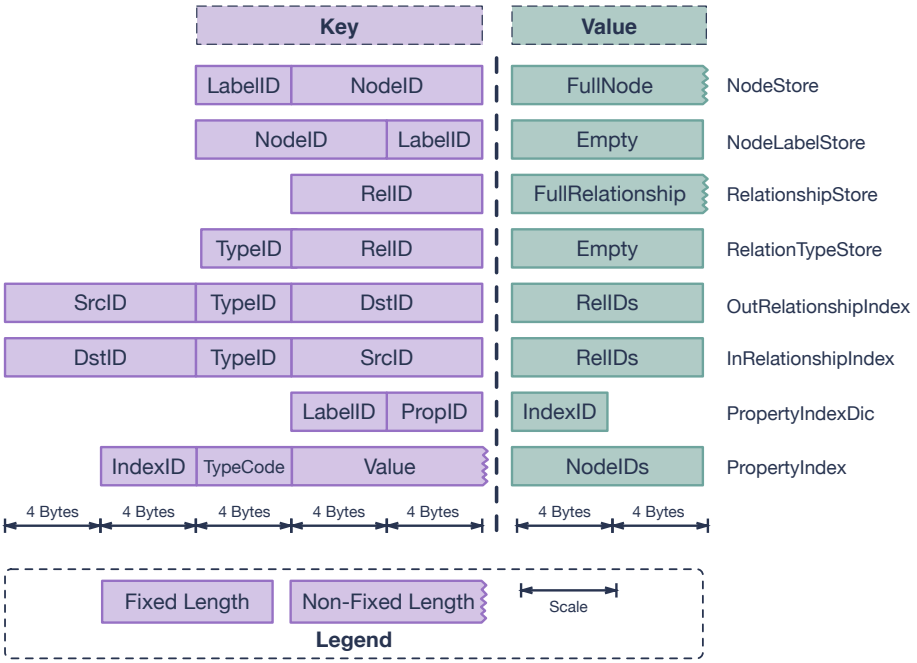

Fig. 1. Mapping Graph Data to Key-Value Storage

- IRI: InRelationIndex, it is the incoming edge index, which modifies the order of the key in the OutRelationIndex to destination node ID, relationship type ID and source node ID, with the rest remaining unchanged.
- PID: PropertyIndexDic, it is the embedded property index dictionary, storing IndexIDs of embedded property indexes. An index is uniquely identified by a LabelID and a PropID (i.e., the ID of the property name). The query engine can determine whether an index exists based on the LabelID and PropID through the PropertyIndexDic; if it exists, further property filtering can be performed in the PropertyIndex (i.e. PI).
- PI: PropertyIndex, it is the embedded property index, used for storing property indexes. In PropertyIndex, the key is a combination of IndexID, TypeCode, and PValue, where IndexID refers to the property index ID described in PropertyIndexDic, TypeCode refers to the type encoding of the property value (such as integers, floating-point numbers, etc.) and PValue refers to the actual value of the property. The value contains the IDs of all nodes with property values equal to PValue under the constraints of the given LabelID and PropID.

2.2 Query

Figure 2 and Figure 3 present three query operations on KVG, namely finding nodes by label, finding nodes by ID, and finding nodes by property. The retrieval operations of relationships share the similar process. More generally, other graph operations can be derived from the steps:

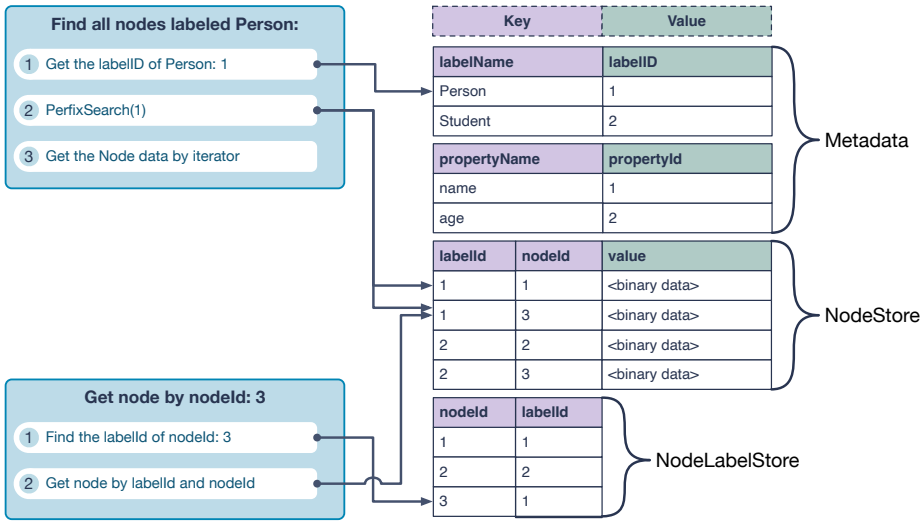

Fig. 2. Find Nodes based on label and ID on KVG

- Find all nodes labeled as *Person*. The process is shown in Figure 2. First, obtain the LabelID of *Person* (assumed to be 1) from the metadata. Then, in the NodeStore, perform a prefix search to find the starting position with LabelID=1. Next, traverse the data downwards until the first node with a different LabelID is encountered.
- Find a node with a specific ID (assumed to be 3). The process is shown in Figure 2. First, in the NodeLabelStore, perform a prefix search to find the first key-value pair with NodeID=3, and get the node's LabelID(1). Then, based on the LabelID(1) and NodeID(3), perform an precise search in the NodeStore to find the corresponding data item and get the full node.
- Filter nodes by the property value, as shown in Figure 3. Suppose the query condition is to find all nodes labeled as *Person* with the *age* value of 31. First, obtain the LabelID(1) and PropID(2) from the metadata. Then, in the PropertyIndexDic, get the indexID(2). Finally, perform a prefix search on the PropertyIndex based on the IndexID(2), TypeCode(2), and PValue(31) to find the corresponding nodeIDs.

3 Implementation and Experiments

We implemented KVG based on RocksDB [4] and named it KVGDB. KVGDB has been open-sourced and adopted as the storage engine of PandaDB [3] [7]. KVGDB adopts Cypher [5] as the query language. We evaluate the performance of KVGDB on the LDBC-SNB dataset. LDBC-SNB [8] is currently the most popular property graph benchmark, which includes a scalable social-network

[3] https://github.com/grapheco/pandadb-v0.3

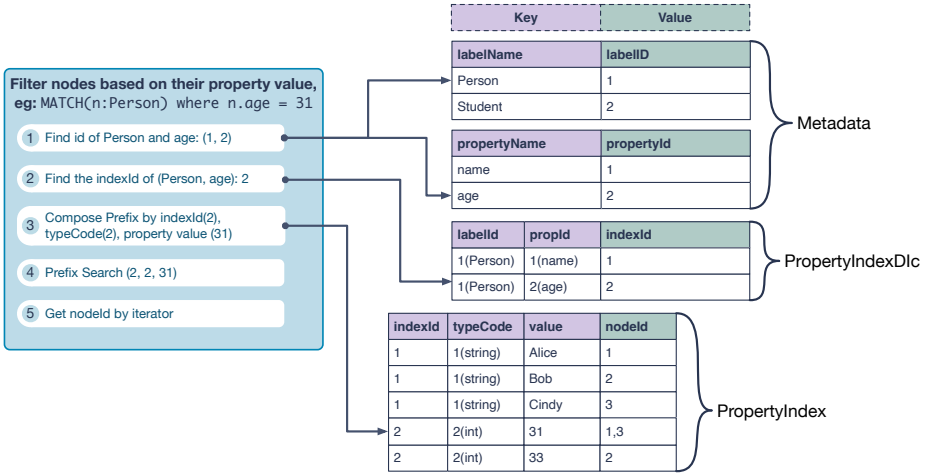

Fig. 3. Filtering Nodes Based on Property on KVG

dataset. The datasets used in this study are detailed in Table 1. The experiment is carried out on a server with 384GB memory, 28 CPU-cores and 10TB hard disks.

Table 1. Details of Dataset

Dataset	Num of nodes	Num of edges	Size on the disk
D1	83,298,515	507,720,806	38GB
D2	2,523,446,454	17,016,067,035	1.24TB

Table 2 lists the basic graph query operations tested in this experiment and their execution times on different datasets. In the table, *KVGDB on D1* and *KVGDB on D2* represent the cost times for KVGDB to execute operations on datasets D1 and D2 (see table 1), respectively. The *Baseline* represents the cost time for Neo4j-community-3.5.6 (one of the most successful graph databases) to execute the queries on dataset D1. We did not evaluate Neo4j on D2, because Neo4j failed to load D2 within 12 hours.

The experimental results show that KVGDB performs better than the baseline, the execution time of most operations is within 10ms, and the execution time of each operation on the two datasets are quite similar. This indicates that the operation time does not increase significantly with the growth of data size, which is consistent with the characteristics and design expectations of KV databases. Notably, according to the data in the first row of the table, the execution time for obtaining all nodes (getAllNodes) and all relationships (getAllRelationships) is much higher than that for other operations. This is because the operation to retrieve all nodes requires deserialization of all node data (and similarly for relationships), making it a traversal operation. The execution time is already close to the limit under existing hardware conditions.

Table 2. Cost Time of Graph Operation on the KVGDB

Operation	Baseline on D1	KVGDB on D1	KVGDB on D2	Operation	Baseline on D1	KVGDB on D1	KVGDB on D2
getAllNodes	396s	36.8s	1225s	getAllRelationships	1218s	123s	6972s
allLabels	3ms	12ms	15ms	getRelationType	9ms	<1ms	<1ms
addLabel	9ms	4ms	4ms	addRelationType	15ms	4ms	21ms
allPropertyKeys	1ms	<1ms	<1ms	allPropertyKeys	1ms	<1ms	<1ms
getPropertyKey	8ms	<1ms	<1ms	getPropertyKey	10ms	<1ms	<1ms
addPropertyKey	11ms	<1ms	2ms	addPropertyKey	7ms	<1ms	<1ms
getNodeById	8ms	6ms	7ms	getRelationById	7ms	<1ms	12ms
hasLabels	15ms	<1ms	¡1m	relSetProperty	7ms	16ms	88ms
nodeAddLabel	12ms	2ms	6ms	relRemoveProperty	8ms	3ms	13ms
nodeRemoveLabel	9ms	6ms	31ms	findToNodeId	7ms	<1ms	<1ms
nodeSetProperty	11ms	4ms	7ms	findFromNodeId	7ms	<1ms	8ms
nodeRemoveProperty	9ms	5ms	8ms	addRelation	10ms	8ms	9ms
addNode	34ms	<1ms	2ms	deleteRelation	12ms	3ms	2ms
deleteNode	16ms	<1ms	2ms	findOutRelations	145ms	3ms	2ms
				findInRelations	901ms	3ms	5ms

4 Conclusion

In this paper, we proposed a method for mapping graph data to key-value storage and implemented a scalable graph database based on RocksDB, namely KVGDB. It can manage data in an embeded fashion and be easily expanded to distributed environments for large-scale datasets. In the future, we will study graph pattern matching algorithms suitable for the features of KVGDB.

References

1. Bebee, B.R., Choi, D., Gupta, A., Gutmans, A., Khandelwal, A., Kiran, Y., Mallidi, S., McGaughy, B., Personick, M., Rajan, K., et al.: Amazon neptune: Graph data management in the cloud. In: ISWC (P&D/Industry/BlueSky) (2018)
2. Bollacker, K., Evans, C., Paritosh, P., Sturge, T., Taylor, J.: Freebase: a collaboratively created graph database for structuring human knowledge. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. pp. 1247–1250 (2008)
3. Deutsch, A., Xu, Y., Wu, M., Lee, V.: Tigergraph: A native mpp graph database. arXiv preprint arXiv:1901.08248 (2019)
4. Dong, S., Kryczka, A., Jin, Y., Stumm, M.: Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. ACM Transactions on Storage (TOS) **17**(4), 1–32 (2021)
5. Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: Cypher: An evolving query language for property graphs. In: Proceedings of the 2018 international conference on management of data. pp. 1433–1445 (2018)
6. Myers, S.A., Sharma, A., Gupta, P., Lin, J.: Information network or social network? the structure of the twitter follow graph. In: Proceedings of the 23rd International Conference on World Wide Web. pp. 493–498 (2014)
7. Shen, Z., Zhao, Z., Wang, H., Liu, Z., Hu, C., Zhou, C.: `PandaDB`: Intelligent management system for heterogeneous data. Int. J. Softw. Informatics **11**(1), 69–90 (2021)
8. Szárnyas, G., Waudby, J., Steer, B.A., Szakállas, D., Birler, A., Wu, M., Zhang, Y., Boncz, P.: The ldbc social network benchmark: Business intelligence workload. Proceedings of the VLDB Endowment **16**(4), 877–890 (2022)