Towards a Unified Storage Scheme for Dual Data Models of Knowledge Graphs

Yuzhou Qin, Xin Wang(✉), and Wenqi Hao

College of Intelligence and Computing, Tianjin University, Tianjin, China
{yuzhou_qin,wangx,haowenqi}@tju.edu.cn

Abstract. As an important cornerstone of artificial intelligence, the knowledge graph is one of the indispensable foundations of the new generation of artificial intelligence from perception to cognition. RDF graphs and property graphs are the two main data models of KGs, and various data management methods have been developed for the two models. However, differences in data models will lead to differences in how the data is stored and manipulated, which will further hinder the widespread application of knowledge graphs. In this paper, we propose a novel unified storage scheme, UniS, which considers the characteristics of the two data model comprehensively. Unlike the existing approaches, the detailed conversion process for different storage forms of data that we devised will make it easier to manage multiple KGs in one database. Meanwhile, the experimental results show that UniS improves the storage and query efficiency by up to an order of magnitude than the state-of-the-art storage engines.

Keywords: Knowledge graphs · Data models · Unified storage scheme

1 Introduction

With the growing application of knowledge graphs in diverse domains, the scale of knowledge graphs (KGs) is dramatically increasing. As the two dominant data models for KGs, RDF (Resource Description Framework) [1] graph and property graph are utilized by most graph databases. For one thing, RDF, which has become a recommended standard for the representation of knowledge graphs by the World Wide Web Consortium (W3C), is widely adopted by databases represented by gStore [2] and Virtuoso [3]. For another, property graphs have been widely used as the data model by graph databases such as JanusGraph [4] and HugeGraph [5].

In recent years, there has been a consensus to unify the data model in the management of knowledge graphs, as different data models will lead to many differences in storage schemes and the databases built on them. Apart from the fact that different data models can cause problems for database users, existing

S. Yang and S. Islam (Eds.): APWeb-WAIM 2022 Workshops, CCIS 1784, pp. 34–44, 2023.
https://doi.org/10.1007/978-981-99-1354-1_4

storage schemes also have problems such as excessive storage overhead and null values. Therefore, we propose a unified storage scheme for RDF graphs and property graphs, which can accommodate both of them. The contributions of this paper can be summarized as follows:

(1) In order to store RDF graphs and property graphs, a unified storage scheme is proposed, i.e., UniS, which considers the characteristics of the two data model comprehensively. With UniS, the entities and edges are clustered and stored in separate tables according to their types, managing data in a unified way.
(2) To manage multiple KGs in one database, we have designed a detailed conversion process for different storage forms of data, which is convenient for further operation of data on this basis and meeting the storage and query load requirements of KGs.
(3) Extensive experiments are carried out to verify the effectiveness and efficiency of UniS. The experimental results show that UniS outperforms the state-of-the-art methods in terms of storage overhead and query overhead.

2 Related Work

In this section, we discuss the related works, including storage schemes and graph databases.

2.1 Storage Schemes

Most of the current storage solutions for RDF and property graphs are relationship-based. Triple table, adopted by 3store [6], stores data into tables with a 3-column structure, where each column corresponds to the subject, predicate, and object of a triple, respectively. Developed from triple table, horizontal table and property table are implemented in DLDB [7] and Jena [8], but they can lead to problems with excessive number of tables and null values. To reduce the join overhead of tables, the system represented by Hexastore [9] adopts sextuple indexing, but it will increase the amount of space required significantly. For the storage of property graph, storage schemes in the form of native graphs or key-value pairs are also utilized. Therefore, it is necessary to develop a unified and efficient storage scheme for RDF and property graphs.

2.2 Graph Database

gStore is an RDF graph database developed for storing triples of data, which utilizes the signature graph corresponding to RDF data and builds VS tree indexes to speed up SPARQL query processing. Moreover, supporting multiple data models including RDF data, Virtuoso is a hybrid database management system with built-in SPARQL and inference. On the other hand, designed for storing property graph, both JanusGraph and HugeGraph are also compatible with the query language Gremlin [10], but the Gremlin implementation cannot migrate directly from JanusGraph to HugeGraph due to the limitations of edge labels on HugeGraph.

3 Preliminaries

In this section, we introduce the definitions of relevant background knowledge.

Definition 1 (RDF Graph). *Let U and L be the disjoint infinite sets of URIs and literals. Then, a tuple t in the form of $\langle s, p, o\rangle \in U \times U \times (U \cup L)$ is called an RDF triple, where s is the subject, p the predicate, and o the object. An RDF dataset T, which is a finite set of triple t, can be converted to an RDF Graph $G = (V, E, \Sigma)$. The V, E, and Σ denote the set of vertices, edges, and edge labels in G, respectively. Formally, $V = \{s \mid (s, p, o) \in T\} \cup \{o \mid (s, p, o) \in T\}$, $E = \{(s, o) \mid (s, p, o) \in T\}$ and $\Sigma = \{p \mid (s, p, o) \in T\}$.*

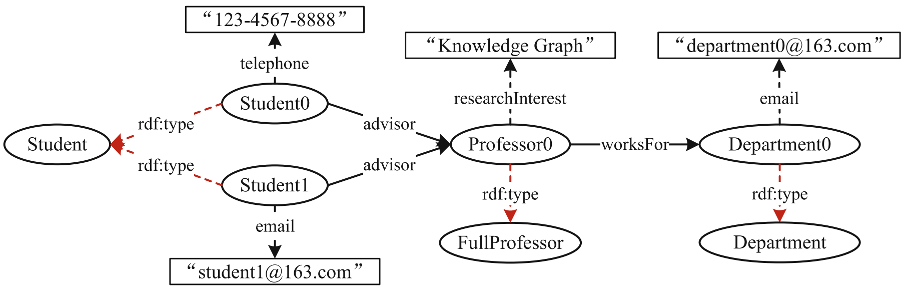


Fig. 1. An RDF graph example

Example 1. An example RDF graph G is shown in Fig. 1, which is composed of resources, associated properties and the relationships between resources. The ellipses and rectangles are used to denote resources and literals, respectively, while directed edges connecting vertices, which corresponds to the triples in the RDF dataset, represent relationships between vertices. In particular, the edge label `rdf:type` is employed to specify the type to which the resource belongs. For instance, the triple (`Professor0`, `rdf:type`, `FullProfessor`) indicates that the type of `Professor0` is `FullProfessor`.

Definition 2 (Property Graph). *Given a property graph $G = (V, E, \eta, src, tgt, \lambda, \gamma)$, where V, E represent the finite set of vertices and edges respectively, and $V \cap E = \emptyset$. The function $\eta : E \to (V \times V)$ denotes the mapping of edge to vertex pair, e.g., $\eta(e) = (v_1, v_2)$ means there is a directed edge e between vertex v_1 and vertex v_2. Moreover, the function $src : E \to V$ and $tgt : E \to V$ denote the mapping of edges to starting vertices and ending vertices, respectively, e.g., $src(e) = v_1$ denotes the starting vertex of edge e is v_1 and $tgt(e) = v_2$ denotes the ending vertex of edge e is v_2. Furthermore, The function $\lambda : (V \cup E) \to Lab$ represents the mapping of vertices or edges to labels, where Lab denotes the set of labels, e.g., let $v \in V$ (or $e \in E$) and $\lambda(v) = l$ (or $\lambda(e) = l$), then l is the label of vertex v (or edge e). In addition, the function $\gamma : (V \cup E) \times K \to Val$ represents the mapping of the associated property to a vertex or an edge, where K is the* *set properties and Val is the set of values, e.g., $v \in V$ (or $e \in E$), $pro \in K$ and $\gamma(v, pro) = val$ (or $\gamma(e, pro) = val$) denotes the value of the property pro on vertex v (or edge e) is val.*

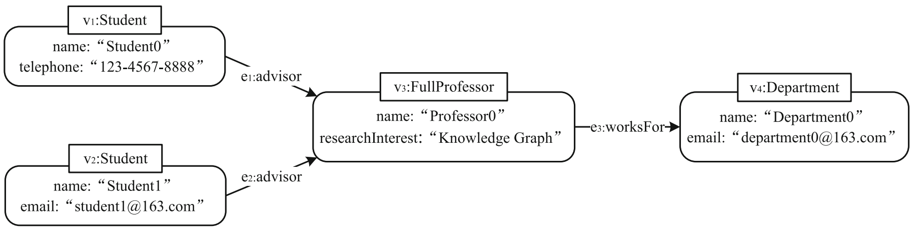


Fig. 2. A property graph example

Example 2. As shown in Fig. 2, every vertice and edge in the property graph has a unique id, and both vertices and edges have labels, e.g., the label `Student` on vertex v1 represents that the type of `Student0` is `Student`. Furthermore, Both vertices and edges have attributes, each of which consists of a key-value pair of an attribute name and an attribute value. For example, the attribute `researchInterest` on vertex v_3 is `Knowledge Graph`.

4 Unified Storage Scheme

In this section, we present a unified data model, named UniS, which is capable of storing both RDF and property graphs. We first introduce the design of the UniS storage model, followed by a description of the process for transforming RDF and property graphs into the UniS format.

4.1 Unified Storage Model

To store RDF graphs and property graphs in a unified manner, we propose a unified data model, which is combined with the characteristics of both the two data models.

UniS is based on the relation model, which is a tuple (R^v, R^e, N, μ), where R^v and R^e represent the set of `entities` and `edges` tables, respectively. For each entity table $r^v \in R^v$, r^v consists of two columns, where the first column records the globally unique identifier of each entity and the second column records the properties of entities. Meanwhile, for edge table $r^e \in R^e$, r^e contains three columns, storing the identifier of source entities, target entities, and properties of each edge, respectively. $N = \{n_1, n_2, ..., n_k\}$ is the set of table names, and the function $\mu : R^v \cup R^e \rightarrow N$ maps relation table r to its corresponding name.

As shown in Fig. 3, UniS is compatible with both property and RDF graphs. For the RDF graphs, we divide the entities into different entity tables based on

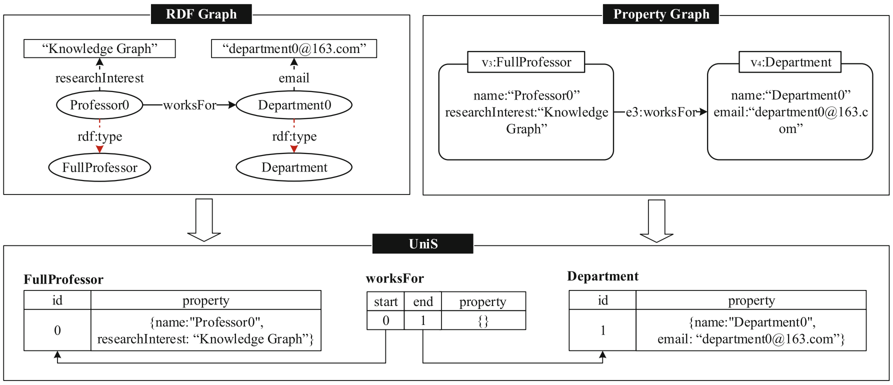


Fig. 3. The storage schema of UniS

the types specified by `rdf:type` and store the constant properties corresponding to the entities in the property column. The RDF graph in Fig. 3 contains two entities, i.e., `Professor0` and `Department0`, whose types are `FullProfessor` and `Department`, respectively, so we create the entity tables `FullProfessor` and `Department` in UniS, and then insert the entity identifiers and properties into them. For the edge `worksFor` between two entities, we create the edge table `worksFor` in UniS for it and insert the identifiers of source entity and target entity, and the properties of the edge into the edge table. For property graphs, similar approach can be adopted to transform them into the UniS. The details of the transformation process will be discussed in Sect. 4.2.

4.2 The Process of Transformation

To accommodate both RDF graphs and property graphs, the UniS are proposed to store both of them. However, due to the differences in the data model between RDF and property graph, various processes need to be designed to transform them into the unified storage schema.

For the storage of RDF graphs, the set of all triples T can be divided into 3 subsets: T_t, T_p, and T_e, which represent the set of triples related to the types of entities, the properties of entities, and the relationship between the entities, respectively. Formally, $T_t = \{t = (s, p, o) \mid t \in T \land p = \texttt{rdf:type}\}$, $T_p = \{t = (s, p, o) \mid t \in T \land o \in L\}$, and $T_e = \{t = (s, p, o) \mid t \in T \land p \neq \texttt{rdf:type} \land o \notin L\}$. In order to transform the triples to the unified storage model, further processing are necessary for the above sets of triples with the following conversion rules:

1. For any triple $t \in T_t$, the entity s should be put in the entity table whose name is o.
2. For any triple $t \in T_p$, the key-value pair (p, o) is inserted into the property column corresponding to entity s in the entity table.

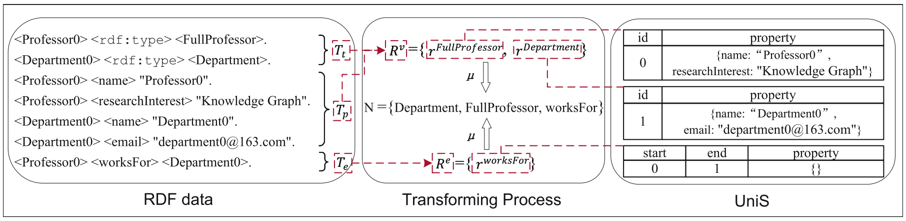
3. For any triple $t \in T_e$, the edge will be put in the edge table named p as a record, where the *start* is s and the *end* is o.

The detailed processing flow is shown in Algorithm 1. The first step is to divide the RDF dataset T into three disjoint subsets T_t, T_p, and T_e (line 1–3). To locate the type and properties of each entity in an efficient way, we first divide the subjects in T_t by `rdf:type` (line 4–5) and the triples in T_p into groups by subject (line 7–8), then assign globally unique identifiers for all subjects in T_t (line 9–10). After that, we employ *props* to record the properties (line 13–15) for each entity and insert (id(s), *props*) into the corresponding entity table (line 16). For T_e, we employ a similar approach to handle it. First, the triples in T_p are grouped by the predicate (line 18–19). Then, for each triple $t = (s, p, o)$ in the group $\mathbb{E}(p)$, $(id(s), id(o), \emptyset)$ will be inserted into the edge table r^e (line 20–22), where $\mu(r^e) = p$.

```
Algorithm 1: Transform RDF Graph
   Input: RDF dataset $T$
   Output: $UniS = (R^v, R^e, N, \mu)$
 1 $T_t \leftarrow \{t = (s,p,o) \mid t \in T \wedge p = \texttt{rdf:type}\}$ ;
 2 $T_p \leftarrow \{t = (s,p,o) \mid t \in T \wedge o \in L\}$;
 3 $T_e \leftarrow \{t = (s,p,o) \mid t \in T \wedge p \neq \texttt{rdf:type} \wedge o \notin L\}$ ;
 4 foreach $t = (s,p,o) \in T_t$ do
 5     $\mathbb{S}(o) \leftarrow \mathbb{S}(o) \cup \{s\}$ ;                    // group subjects by its type
 6     $N \leftarrow N \cup \{o\}$ ;
 7 foreach $t = (s,p,o) \in T_p$ do
 8     $\mathbb{P}(s) \leftarrow \mathbb{P}(s) \cup \{t\}$ ;                      // group triples by subject
 9 foreach $s \in \{s \mid (s,p,o) \in T_t\}$ do
10     $id(s) \leftarrow$ a globally unique identifier ;
11 foreach $n \in N$ do
12     foreach $s \in \mathbb{S}(n)$ do
13         foreach $(s,p,o) \in \mathbb{P}(s)$ do
14             $props(p) \leftarrow o$ ;                  // insert (p,o) into properties
15         $props(“uri“) \leftarrow s$ ;
16         $r^v \leftarrow r^v \cup \{(id(s), props)\}$ ;      // insert data into entity table
17     $R^v \leftarrow R^v \cup \{r^v\}$; $\mu(r^v) \leftarrow n$;
18 foreach $t = (s,p,o) \in T_e$ do
19     $\mathbb{E}(p) \leftarrow \mathbb{E}(p) \cup \{t\}$ ;             // group triples in $T_e$ by predicate
20 foreach $p \in \{p \mid (s,p,o) \in T_e\}$ do
21     foreach $t = (s,p,o) \in \mathbb{E}(p)$ do
22         $r^e \leftarrow r^e \cup \{(id(s), id(o), \emptyset)\}$ ;   // insert edge data into edge table
23     $R^e \leftarrow R^e \cup \{r^e\}$; $\mu(r^e) \leftarrow p$ ; $N \leftarrow N \cup \{p\}$;
24 return $(R^v, R^e, N, \mu)$ ;
```

The time complexity of Algorithm 1 is $O(|T| \cdot \log(|T|))$, where $|T|$ is the numbers of triples in RDF dataset. The time complexity of the algorithm consists of two parts: (1) traverse triples in T_t, T_p, and T_e to generate mappings $\mathbb{S}$, $\mathbb{P}$, and $\mathbb{E}$, respectively, with complexity of $O(|T| \cdot \log(|T|))$; (2) traverse triples in $\mathbb{S}$ and $\mathbb{E}$ to insert data into corresponding entity or edge tables, with complexity of $O(|T|)$. Hence, the overall time complexity of this algorithm is $O(|T| \cdot \log(|T|))$.

For the storage of property graphs, it is relatively easy to transform them to the unified storage model as the property graph provides built-in support for properties on vertices and edges. Specifically, the data of vertices or edges with different labels in the property graph will be converted to records in the corresponding entity tables or edge tables, then their properties are stored in the properties columns. For instance, let $\lambda(v) = l$ (or $\lambda(e) = l$) and $\gamma(v, pro) = val$ (or $\gamma(e, pro) = val$), the entity v (or the edge e) should be inserted into the entity table (or the edge table) named l, and the (pro, val) will be put in the property column corresponding to entity v (or the edge e).



<Professor0> <rdf:type> <FullProfessor>.
<Department0> <rdf:type> <Department>.
<Professor0> <name> "Professor0".
<Professor0> <researchInterest> "Knowledge Graph".
<Department0> <name> "Department0".
<Department0> <email> "department0@163.com".
<Professor0> <worksFor> <Department0>.
T_t
T_p
T_e
$R^v = \{r^{FullProfessor}, r^{Department}\}$
μ
N = {Department, FullProfessor, worksFor}
μ
$R^e = \{r^{worksFor}\}$
id
property
0
{name: "Professor0", researchInterest: "Knowledge Graph"}
id
property
1
{name: "Department0", email: "department0@163.com"}
start
end
property
0
1
{}
RDF data
Transforming Process
UniS

Fig. 4. The process of UniS

Example 3. Figure 4 shows the process of transforming RDF data to UniS. The set of triples is first classified according to the structure of them, then R^v can be obtained by further processing T_t and T_p, while R^e can be constructed according to T_e. Specifically, based on the type and attribute information of `Professor0`, the entity table $r^{FullProfessor}$ can be constructed. Moreover, the type `FullProfessor` is added to the set N and the mapping function μ between $r^{FullProfessor}$ and `FullProfessor` is built. In the same way, we can construct the entity table $r^{Department}$ and the edge table $r^{worksFor}$, and further obtain the final result of UniS.

5 Experiments

In this section, to verify the efficiency of the unified storage scheme, i.e., UniS, we compare it against the RDF databases gStore [2], Virtuoso [3] and the property graph database JanusGraph, HugeGraph on different datasets.

5.1 Experimental Settings

On the top of Nebula Graph, the proposed unified storage scheme is implemented and deployed on a 4-node cluster, which has a 16-core Intel(R) Xeon(R) Silver

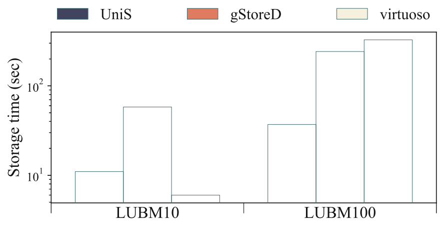
4216 2.10 GHz CPU, 512 GB of RAM, and 1.92 TB SSD, running the 64-bit CentOS 7.7 operating system.

Data sets. Our experiments are conducted on the datasets of LUBM [11] and LDBC-SNB [12]. The LDBC Social Network Benchmark (SNB) models the social network graph, including people and their activities over time. The Lehigh University Benchmark (LUBM) is developed for evaluating the performance of Semantic Web repositories, which describes universities, departments, and their activities. In our experiment, we generate several different scales of data for both datasets.

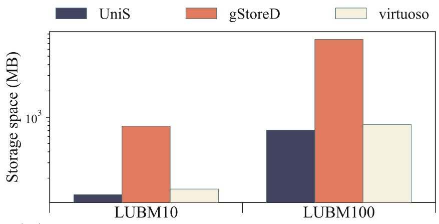
Baselines. To verify the efficiency of storage, we compare the unified storage scheme with gStore, Virtuoso, JanusGraph, and HugeGraph in terms of both storage space overhead and loading time. Specifically, gStore 0.3.0 and Virtuoso 7.2.6 are RDF graph databases and JanusGraph 0.5.3 and HugeGraph 0.11.2 are property graph databases. Moreover, based on the unified storage scheme, experiments of query test was carried out to verify the query efficiency.

5.2 Experimental Results

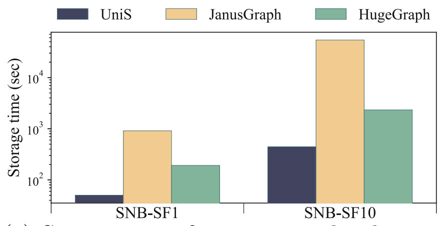
Exp 1. Storage Efficiency. As shown in Fig. 5(a) and Fig. 5(b), UniS is significantly more efficient than gStoreD in terms of storage time and space for the LUBM dataset. UniS has a similar storage space overhead compared to Virtuoso, and the storage time of UniS is longer than Virtuoso when the data volume is small. However, as the data volume increases, UniS outperforms Virtuoso by up to an order of magnitude in loading data. For the LDBC dataset, as shown in Fig. 5(c) and Fig. 5(d), the storage efficiency of UniS is better than JanusGraph and HugeGraph.



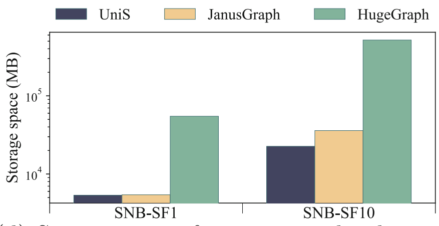
(a) Storage Time for RDF databases



(b) Storage space for RDF databases



(c) Storage time for property databases



(d) Storage space for property databases

Fig. 5. The experimental results of storage efficiency

There are two reasons for these results: (1) we devise an efficient transformation process where the time complexity is $O(|T| \cdot \log(|T|))$, which improves the efficiency of data loading significantly; (2)We employee advanced compression techniques, including dictionary encoding and common prefix extraction, to optimize the storage of large RDF datasets in UniS. By implementing these methods, we're able to compress the raw data and save on storage space, thus enabling UniS to store more data without requiring additional storage resources.

Exp 2. Query Efficiency. We executed the eight queries[1] over the LUMB10 dataset to verify the query efficiency of UniS on the RDF dataset. As can be seen in Fig. 6, the average query efficiency of UniS is 4.26 and 8.35 times higher than that of gStoreD and Virtuoso, respectively, for the following reasons: (1) UniS stores data of different types separately, which significantly accelerate the data filtering for queries of specified types. (2) UniS stores entities and their properties together, eliminating several join operations and improving the performance of queries about multiple properties of a single entity, such as Q6. (3) UniS compress the raw data, improving query efficiency by alleviating the burden of the disk.

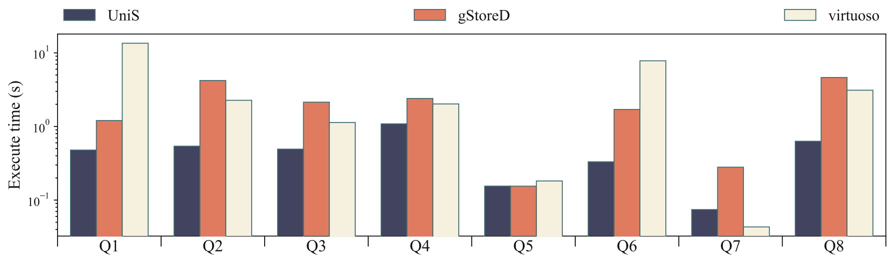


Fig. 6. Execute time on LUBM10

As shown in Fig. 7, we executed seven interactive short queries provided by LDBC [12] over the LDBC-SNB SF1 dataset to verify the query efficiency of UniS on the property graphs. It can be seen that the average query efficiency of UniS is 1.28 and 1.75 times that of JanusGraph and HugeGraph, respectively, except for the queries that do not finish. The most significant advantage of UniS compared to other property databases is that it exploits dictionary encoding and prefix extraction to compress the raw data to reduce the cost of disk read, thus improving the query performance.

[1] https://github.com/rainboat2/KGMA2022.git.

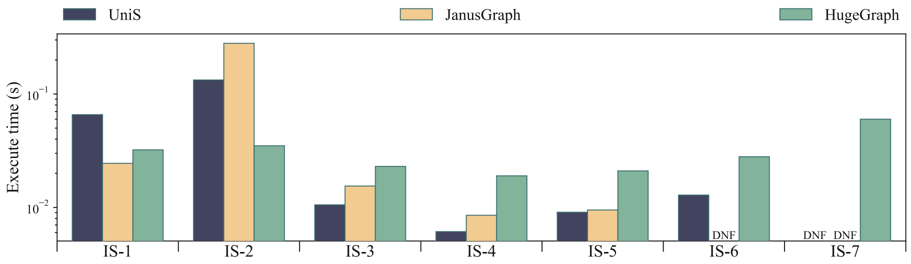


Fig. 7. Execute time on LDBC-SNB SF1 (DNF denotes dooes not finish within 30 min.)

6 Conclusion

In this paper, we propose UniS, a unified storage scheme for different data models on knowledge graphs. Considering the characteristics of RDF graphs and the property graphs comprehensively, the unified storage model is utilized to store the two data models in a unifed way. Furthermore, a detailed conversion process is devised, which makes it easier to manage multiple KGs in one database. Micro-benchmarks over LUBM and LDBC are proposed to verify the efficiency and effectiveness of UniS. The experimental results show that UniS outperforms the state-of-the-art methods in terms of storage overhead and query overhead by up to an order of magnitude.

Acknowledgments. This work is supported by the National Key Research and Development Program of China (2019YFE0198600) and National Natural Science Foundation of China (61972275).

References

1. Consortium, W.W.W., et al.: Rdf 1.1 concepts and abstract syntax (2014)
2. Das, S., Agrawal, D., El Abbadi, A.: G-store: a scalable data store for transactional multi key access in the cloud. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 163–174 (2010)
3. Erling, O., Mikhailov, I.: Rdf support in the virtuoso dbms. In: Pellegrini, T., Auer, S., Tochtermann, K., Schaffert, S. (eds.) Networked Knowledge - Networked Media. Studies in Computational Intelligence, vol. 221. Springer, Berlin. pp. 7–24 (2009) https://doi.org/10.1007/978-3-642-02184-8_2
4. Authors, J.: Janusgraph–distributed graph database. http://janusgraph.org/ (2020)
5. Team, T.H.: The hugegraph manual. https://hugegraph.github.io/hugegraph-doc/ (2020)
6. Harris, S., Gibbins, N.: 3store: Efficient bulk rdf storage (2003)
7. Pan, Z., Heflin, J.: Dldb: extending relational databases to support semantic web queries. Lehigh univ bethlehem pa dept of computer science and electrical engineering, Technical Report (2004)

8. Wilkinson, K., Wilkinson, K.: Jena property table implementation (2006)
9. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proc. VLDB Endowment **1**(1), 1008–1019 (2008)
10. TinkerPop, A.: Tinkerpop3 documentation v.3.3.3. http://tinkerpop.apache.org/docs/3.3.3/reference/ (2018)
11. Guo, Y., Pan, Z., Heflin, J.: Lubm: a benchmark for owl knowledge base systems. J. Web Seman. **3**(2–3), 158–182 (2005)
12. Angles, R., et al.: The ldbc social network benchmark. arXiv preprint arXiv:2001.02299 (2020)