An Ontology-Aware Unified Storage Scheme for Knowledge Graphs

Sizhuo Li[1], Guozheng Rao[1,2(✉)], Baozhu Liu[1], Pengkai Liu[1], Sicong Dong[1], and Zhiyong Feng[1,2]

[1] College of Intelligence and Computing, Tianjin University, Tianjin, China
{lszskye,rgz,liubaozhu,liupengkai,sicongdong,zyfeng}@tju.edu.cn
[2] Tianjin Key Laboratory of Cognitive Computing and Application, Tianjin, China

Abstract. With the development of knowledge-based artificial intelligence, the scale of knowledge graphs has been increasing rapidly. The RDF graph and the property graph are two mainstream data models of knowledge graphs. On the one hand, with the development of the Semantic Web, there are a large number of RDF knowledge graphs. On the other hand, property graphs are widely used in the graph database community. However, different families of data management methods of RDF graphs and property graphs have been seperately developed in each community over a decade, which hinder the interoperability in managing large knowledge graph data. To address this problem, we propose a unified storage scheme for knowledge graphs which can seamlessly accommodate both RDF and property graphs. Meanwhile, the concept of ontology is introduced to meet the need for RDF graph data storage and query load. Experimental results on the benchmark datasets show that the proposed ontology-aware unified storage scheme can effectively manage large-scale knowledge graphs and significantly avoid data redundancy.

Keywords: Knowledge graph · Unified storage scheme · Ontology-aware

1 Introduction

Knowledge graphs have become the cornerstone of artificial intelligence. With the applications of artificial intelligence, more and more fields begin to organize and publish their domain knowledge in the form of knowledge graphs. Knowledge graphs can not only describe various entities and concepts which exist in the real world, but also can depict the relationships between these entities and concepts. At present, knowledge graphs have been widely used in the fields of big data analysis [1], knowledge fusion [2], precision marketing [3], and semantic search [4].

As the demand of knowledge-based AI applications, the amount of knowledge graph data has been dramatically increasing. Currently, it is common that knowledge graphs have millions of vertices and billions of edges. Many knowledge graphs in the LOD (Linked Open Data) cloud diagram have more than 1 billion

X. Wang et al. (Eds.): APWeb-WAIM 2020, LNCS 12317, pp. 212–226, 2020.
https://doi.org/10.1007/978-3-030-60259-8_17

triples. For example, the number of triples of the latest version of the DBpedia [5] dataset has reached 13 billion. Meanwhile, a great amount of graph data has been stored as property graphs. Therefore, many systems are developed in graph database industry, including Neo4j [6], TigerGraph [7], and OrientDB [8].

In order to manage large-scale knowledge graph data, two mainstream data models of knowledge graphs have been developed: the RDF (Resource Description Framework) model [9] and the property graph model. RDF is a standard data model developed by the World Wide Web Consortium to represent and manage information on the Semantic Web. In the graph database community, the property graph model is another common data model which has built-in support for vertex and edge properties [10]. At present, these two mainstream data models for knowledge graphs have not been unified in a broader perspective of knowledge graph data management, which hinder the interaction while managing knowledge graphs from different communities. A unified data model helps reduce the cost of development of database management systems and realize the interoperability of different types of knowledge graphs at the same time. It has become an urgent need that RDF and property graphs can be effectively managed in a unified storage scheme.

In this paper, we present a unified storage scheme for both RDF and property graphs. Considering the mature storage management facilities in relational databases, our unified storage scheme for knowledge graphs has been implemented based on an RDBMS, using the relational data model as the physical layer to realize our knowledge graph data model. Since knowledge graphs support querying instance information with rich ontology semantic information, it is necessary to propose an ontology-aware unified storage scheme for knowledge graphs. Ontology is introduced to optimize storage and facilitate query as a heuristic information. Finally, we have designed and realized a prototype system that implements the proposed storage scheme and supports efficient query processing.

Our contributions can be summarized as follows:

1) We propose a novel unified storage scheme for knowledge graphs based on relational data model, which can seamlessly accommodate both RDF and property graphs.
2) We introduce ontology as a rich semantic information to optimize our knowledge graph storage scheme. Additionally, the prefix encoding is adopted to save storage space and reflect the hierarchical information between the ontologies.
3) Extensive experiments on several datasets are conducted to verify the effectiveness and efficiency of our storage scheme. The experimental results show that the triple traversal time of our scheme is less than that of the Neo4j.

The rest of this paper is organized as follows. Section 2 briefly introduces the related work and several formal definitions are given in Sect. 3. The ontology-aware unified storage scheme is described in detail in Sect. 4. Section 5 shows experimental results on benchmark datasets. Finally, we conclude in Sect. 6.

2 Related Work

Relational storage scheme is one of the main methods of storing knowledge graph data. In this section, various relational storage schemes are introduced, including Triple table, Horizontal table, Property table, Vertical partitioning, and Sextuple indexing.

Triple Table. Triple table is a storage scheme with a three-column table in a relational database. The scheme of this table is:

$$triple\_table(subject, predicate, object)$$

Each triple in a knowledge graph is stored as a row in *triple_table*. The triple table is the simplest way to store knowledge graphs in a relational database. Although the triple table storage scheme is clear, the number of rows in the *triple_table* is equal to the number of edges in the corresponding knowledge graph. Therefore, there will be many self-joins after translating a knowledge graph query into an SQL query. The representative system adopting triple table storage scheme is 3store [11].

Horizontal Table. Each row of the horizontal table stores all the predicates and objects of a subject. The number of rows is equal to the number of different subjects in the corresponding knowledge graph. The horizontal table storage scheme, however, is limited by the following disadvantages: (1) the structure of the horizontal table is not stable. The addition, modification or deletion of predicates in knowledge graphs will directly result in the addition, modification or deletion of columns in the horizontal table. The change to the structure of the table often leads to high costs; and (2) the number of columns in the horizontal table is equal to the number of different predicates in the corresponding knowledge graph. In a real-world large-scale knowledge graph, the number of predicates is likely to reach tens of thousands, which is likely to exceed the maximum number of columns in the table allowed by the relational database. The representative system adopting horizontal table storage scheme is DLDB [12].

Property Table. The property table is a refinement of the horizontal table, storing those subjects of the same type in one table, which solves the problem of exceeding the limit of the maximum number of columns in the horizontal table scheme. However, there still exists several drawbacks of the property table storage scheme: (1) the number of predicates is likely to reach tens of thousands in a real-world large-scale knowledge graph and thus a large number of tables need to be created. The number of the tables may exceed the maximum number of tables; and (2) the property table storage scheme can cause the problem of null value. The representative system adopting property table storage scheme is Jena [13].

Vertical Partitioning. The vertical partitioning storage scheme creates a two-column table for each predicate [14]. Subjects and objects of the same predicate will be stored in one table. Compared with the previous storage schemes, the

problem of null value is solved. However, the vertical partitioning storage scheme also has its limitation: the number of tables to be created is equal to the number of different predicates in knowledge graphs, and the number of predicates in a real-world large-scale knowledge graph may exceed several thousand, which leads to high cost while maintaining the database. The representative database adopting the vertical partitioning storage scheme is SW-Store [15].

Sextuple Indexing. Six tables are built to store all the six permutations of triples in sextuple indexing storage scheme, namely `spo`, `pos`, `osp`, `sop`, `pso`, and `ops`. The sextuple indexing storage scheme helps alleviate the self-join problem of single table and improve the efficiency of some typical knowledge graph queries. The sextuple indexing storage scheme adopts typical "space-for-time" strategy, therefore, a large amount of storage space is required. Typical systems adopting the sextuple indexing storage scheme are RDF-3X [16] and Hexastore [17].

Inspired by the above relational storage schemes, our unified storage scheme adopts the relational data model as the physical layer to realize our knowledge graph data model. The details of our scheme will be introduced in Sect. 4.

3 Preliminaries

In this section, we introduce several basic background definitions, including RDF graph, property graph, triple partition, and triple classification, which are used in our algorithms.

Definition 1 (RDF Graph). *Let U, B, and L be three infinite disjoint sets of URIs, blank nodes, and literals, respectively. A triple* $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$ *is called an RDF triple, where s is the subject, p is the predicate, and o is the object. A finite set of RDF triples is called an RDF graph.*

Definition 2 (Property Graph). *A property graph is a tuple* $G = (V, E, \rho, \lambda, \sigma)$ *where:*

1) *V is a finite set of vertices,*
2) *E is a finite set of edges and* $V \cap E = \emptyset$,
3) $\rho : E \rightarrow (V \times V)$ *is a mapping that associates an edge with a pair of vertices. For example,* $\rho(e) = (v_1, v_2)$ *indicates that e is a directed edge from vertex* v_1 *to* v_2,
4) *Let Lab be the set of labels.* $\lambda : (V \cup E) \rightarrow Lab$ *is a mapping that associates a vertex or an edge with a label, i.e.,* $v \in V$ *(or* $e \in E$*) and* $\lambda(v) = l$ *(or* $\lambda(e) = l$*), then l is the label for vertex v (or edge e),*
5) *Let Prop be the set of properties and Val be the set of values.* $\sigma : (V \cup E) \times Prop \rightarrow Val$ *is a mapping that associates a vertex (or edge) with its corresponding properties, i.e.,* $v \in V$ *(or* $e \in E$*),* $p \in Prop$ *and* $\sigma(v, p) = val$ *(or* $\sigma(e, p) = val$*), then the value of the property p of the vertex v (or edge e) is val.*

Definition 3 (Triple Partition). *Let T be a finite set of RDF triples whose values of subjects and objects are not blank nodes. T can be divided into three subsets, including X(T), Y(T), and Z(T).*

$$X(T) = \{(s, p, o) \mid (s, p, o) \in T \wedge p = \texttt{rdf:type}\} \tag{1}$$

$$Y(T) = \{(s, p, o) \mid (s, p, o) \in T \wedge o \in L\} \tag{2}$$

$$Z(T) = \{(s, p, o) \mid (s, p, o) \in T \wedge p \neq \texttt{rdf:type} \wedge o \notin L\} \tag{3}$$

The three subsets satisfy the following two conditions: (1) $X(T) \cup Y(T) \cup Z(T) = T$; and (2) $X(T) \cap Y(T) \cap Z(T) = \emptyset$. Since a triple set T is divided into three subsets, the triple classification of T based on triple partition can be defined as Definition 4.

Definition 4 (Triple Classification). *Let C be the set of classes of triples. $C = \{mem, prop, edge\}$. $\varphi : T \rightarrow C$ is a mapping that associates an RDF triple with its corresponding class.*

$$\varphi(t) = \begin{cases} mem & if\ t\ \in X(T) \\ prop & if\ t\ \in Y(T) \\ edge & if\ t\ \in Z(T) \end{cases}$$

The example RDF graph shown in Fig. 1 describes a music knowledge graph where $(LangLang,\ plays,\ FateSymphony) \in Z(T)$ and $\varphi((LangLang,\ plays,\ FateSymphony)) = edge$.

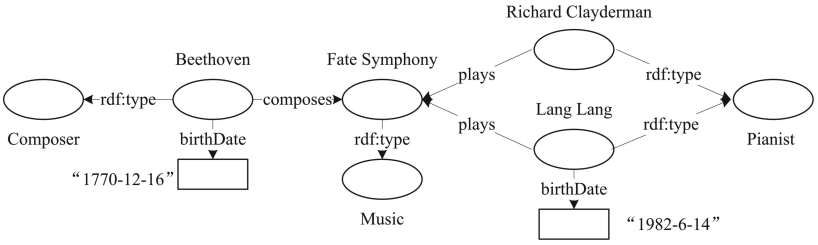


Fig. 1. An example RDF graph.

4 Ontology-Aware Unified Storage Scheme

In this section, we first propose a basic storage scheme for both RDF and property graphs, then we optimize the basic storage scheme by introducing the information of ontology. Finally, we present our data loading algorithm.

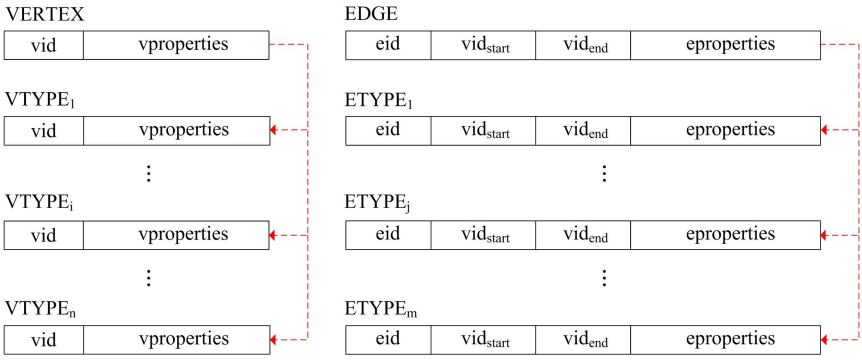


Fig. 2. The basic storage scheme.

4.1 A Basic Storage Scheme

The basic storage scheme is composed of several relations, including `VERTEX`, `EDGE`, $\texttt{VTYPE}_1$, $\texttt{VTYPE}_2$,..., $\texttt{VTYPE}_n$ (n is the number of vertex labels), $\texttt{ETYPE}_1$, $\texttt{ETYPE}_2$,..., $\texttt{ETYPE}_m$ (m is the number of edge labels), as shown in Fig. 2. Actually, relation $\texttt{VTYPE}_i$ is a partition of relation `VERTEX`, while relation $\texttt{ETYPE}_j$ is a partition of relation `EDGE`.

For property graphs, the first column in `VERTEX` records the encodings of all the vertices in the property graph, while the properties of those vertices are kept in the second column. Meanwhile, the first column in `EDGE` holds the encodings of all the edges, while the information of the head vertices, tail vertices, and properties of those edges are kept in the second, third, and forth column, respectively. $\texttt{VTYPE}_i$ $(0 \leqslant i \leqslant n)$ records the information of those vertices of the same label, while $\texttt{ETYPE}_j$ $(0 \leqslant j \leqslant m)$ records the information of those edges of the same label.

For RDF graphs, the first column of each row in `VERTEX` records the URI of an instance. The URIs of properties and the corresponding literals of this instance are kept in the second column. Besides, the first column of each row in `EDGE` holds a predicate p where $\varphi((s, p, o)) = edge$. The subjects and objects of this predicate are stored in the second column and the third column, respectively. $\texttt{VTYPE}_i$ records the information of those vertices of the same type, while $\texttt{ETYPE}_j$ records the information of those edges of the same type.

4.2 The Ontology Information

RDFS (RDF Schema) is an extension of RDF, and it provides the framework to describe application-specific classes and properties. Classes in RDFS are much like classes in object oriented programming languages which allows resources to be defined as instances of classes, and subclasses of classes. More specifically, the class `rdfs:Class` declares a resource as a class for other resources. The property `rdfs:subClassOf` is an instance of `rdf:property` that is used to state that all

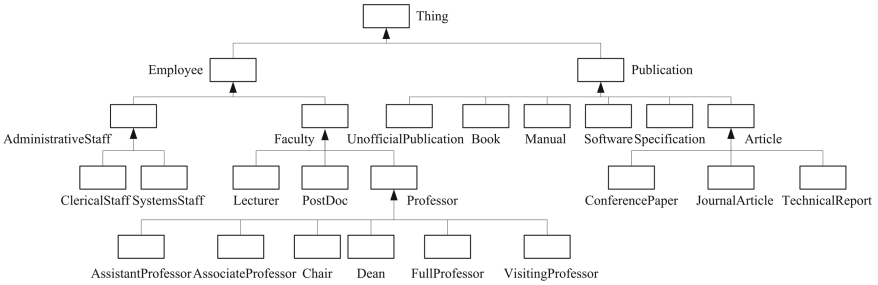


Fig. 3. Ontology hierarchical structure.

the instances of one class are instances of another. Since a class can be further refined into subclasses, an ontology of an RDF graph is actually a hierarchical structure of classes.

Algorithm 1: OntologyEncoding(T)

Input : Ontology $T = (V, E, v_0, Label, Id, \mu)$, where $Id = \emptyset$
Output: The encoded ontology T

```
1  V_l ← getLeaf(T.V);                          // Get all the leaf node in T
2  foreach v ∈ V_l do
3      return id ← getCode(v);
4  return T;
5  Function getCode (v)
6      if v isSubclassOf v_j then
7          v_j.index ← |V_l|;
8          V_l ← V_l ∪ {v_j};
9          return getCode(v_j) + v.index;
           // The plus sign refers to string concatenation
10     else
11         return v.index;
```

The structure of an ontology can be described in terms of a tree, where the nodes represent classes, and the edges represent the relationships between them. The definition of an ontology can be formally given as:

Definition 5 (Ontology). *An ontology is a 6-tuple* $O = (V, E, v_o, Label, Id, \mu)$ *where*

1) V is a finite set of nodes,
2) E is a finite set of edges and $V \cap E = \emptyset$,
3) v_0 is the root node of the tree structure and $v_0 \in V$,
4) Let Label be the set of labels,

5) *Let Id be the set of the encodings of nodes. Each node has a unique encoding, which contains its complete hierarchical information,*
6) $\mu : V \rightarrow Id$ *is a mapping that associates a node with its corresponding encoding, i.e.,* $v \in V$ *and* $\mu(v) = id$*, then the encoding of node* v *is* id*.*

The example ontology hierarchical structure is extracted from Lehigh University Benchmark (LUBM) [18]. The root of the ontology hierarchy is *owl:Thing*, as shown in Fig. 3. Meanwhile, the hierarchical structure is formed by transitivity of the property `rdfs:subClassOf`, e.g., *Publication* has *Article*, *Book*, *Manual*, *Software*, *Specification*, and *UnofficialPublication* as its direct subclasses.

In order to save storage space and reflect the hierarchical structure of the ontology, we leverage the prefix encoding to encode classes in each ontology. Algorithm 1 shows the recursive procedure of ontology encoding, meaning that it first encodes the leaf node, then upward encodes the nodes until the root *owl:Thing* is encountered.

4.3 An Optimized Storage Scheme

In this section, we modify the basic storage scheme and propose an ontology-aware unified storage scheme for knowledge graphs. The optimized storage scheme is composed of three main relations, including `VERTEX`, `EDGE`, and `ONTOLOGY`, and several auxiliary relations, as shown in Fig. 4. `VERTEX` and `EDGE` store the information of all the vertices and edges in a knowledge graph. Meanwhile, the encodings of the ontologies in RDF graphs are kept in relation `ONTOLOGY`.

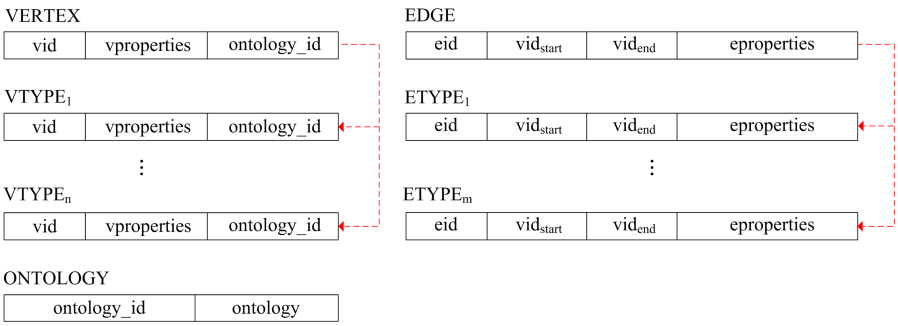


Fig. 4. The ontology-aware unified storage scheme.

The storage scheme of RDF and property graphs in our optimized version is similar to that in the basic version, except for an extra relation `ONTOLOGY` and an extra field *ontology_id* in relation `VERTEX` and relation $\texttt{VTYPE}_i$. Since the prefix encoding is adopted to reflect the hierarchical structure between RDF classes, queries assuming the *subClassOf* relationship between an RDF class and

its subclasses can be completed by using the keyword "`LIKE`" in a fuzzy match inquiry process. Therefore, the ontology helps improve the reasoning capability and optimize queries as a heuristic information.

When storing a property graph, the value of *ontology_id* will be left empty. An example scheme for a given property graph is shown in Fig. 5 (a), and the property graph is shown in Fig. 5 (b).

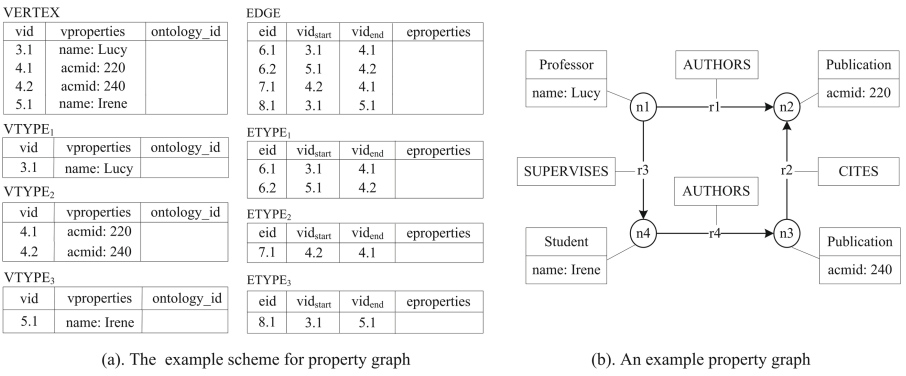


Fig. 5. Example scheme for a given property graph.

When storing an RDF graph, the first column of each row in `VERTEX` records the URI of an instance, while the third column records the encoding of the corresponding ontology. An example scheme for RDF graph is shown in Fig. 6, and the corresponding RDF graph is shown in Fig. 1.

4.4 Loading Algorithm

We present a loading algorithm which is shown in Algorithm 2 to import RDF data. The input of the algorithm is an RDF graph G and its corresponding ontology T. The processed triples are stored into an auxiliary data structure *VEO*. The auxiliary data structure *VEO* is built on top of our storage scheme.

Algorithm 2 consists of three parts: (1) if $\varphi((s,p,o)) = mem$ (lines 2–5), `getLabel` function is invoked to obtain the corresponding label of o, then `traverseO` function is called to traverse T in depth-first order and get the node v_c with label l, finally function μ gets the ontology encoding of v_c; (2) if $\varphi((s,p,o)) = prop$ (lines 6–9), `getProperty` function is provided to get the property information corresponding to s, then *VEO* is updated with the latest value of properties; and (3) if $\varphi((s,p,o)) = edge$ (lines 10–11), (p,s,o) is inserted into *VEO*.

Theorem 1. *Given an RDF graph G and its corresponding ontology T, we assume that the triples in G are processed and stored into the auxiliary data structure VEO by Algorithm 2. The time complexity of Algorithm 2 is bounded*

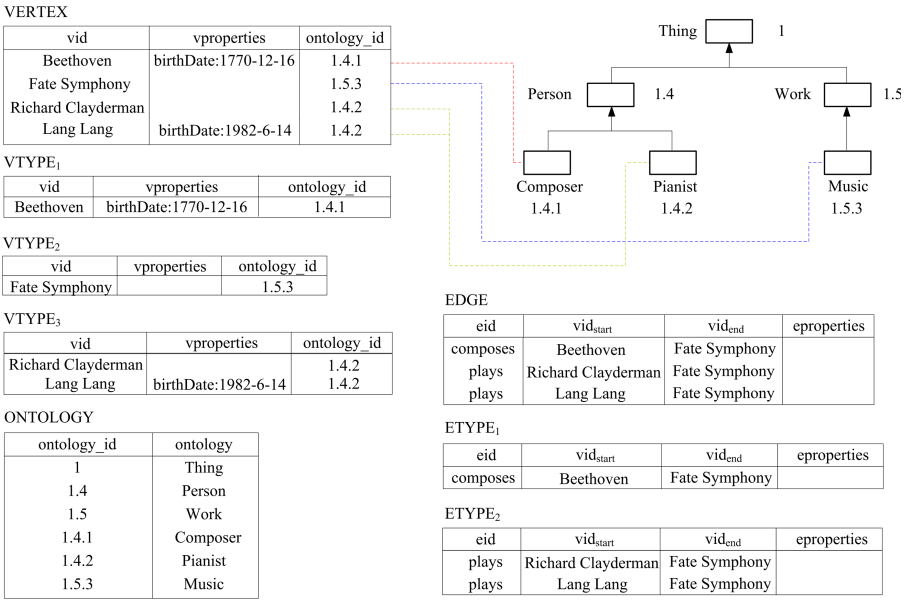


Fig. 6. Example scheme for a given RDF graph.

Algorithm 2: TRIPLELOADING(G, T)

Input : RDF graph G, Ontology $T = (V, E, v_0, Label, Id, \mu)$
Output: The auxiliary data structure VEO

foreach $(s, p, o) \in G$ **do**
  if $\varphi((s, p, o)) = mem$ **then**
    $l \leftarrow$ `getLabel`(o);
    $v_c \leftarrow$ `traverseO`(v_0, l); `// Traverse ontology` T `from` v_0
    insert $(s, \mu(v_c))$ into VEO; `// Get the ontology encoding of` v_c
  else if $\varphi((s, p, o)) = prop$ **then**
    $property \leftarrow$ `getProperty`(VEO, s);
    $p \leftarrow property + p + o$;
    update p in VEO;
  else
    insert (p, s, o) into VEO;
return VEO;
Function `traverseO` (v, l)
  if $v \neq$ NULL **then**
    if $v.lab = l$ **then** `// Depth-first search`
      return v;
    else
      `traverseO`$(v.firstChild, l)$;
      `traverseO`$(v.nextBrother, l)$;

by $O(|S|(|V|+|E|))$, where $|S|$ is the number of triples in G, $|V|$ is the number of nodes in ontology T, and $|E|$ is the number of edges in ontology T.

Proof. (Sketch) Since Algorithm 2 contains three branches and the total time complexity is equal to the time complexity of the branch with the largest time complexity, the time complexity of Algorithm 2 is equal to the complexity of the first branch. In the worst case, every triple in G belongs to class *mem*. The time complexity of traversing T is $|V|+|E|$. Thus, the time complexity of Algorithm 2 is bounded by $O(|S|(|V|+|E|))$ □

5 Experiments

In this section, extensive experiments were conducted to evaluate the performance of our scheme, using open source database AgensGraph [19] as our relational backend. Experiments were carried out on a machine which has 4-core, Intel(R) Xeon(R) Platinum 8255C CPU @ 2.50 GHz system, with 16 GB of memory, running 64-bit CentOS 7.

5.1 Datasets

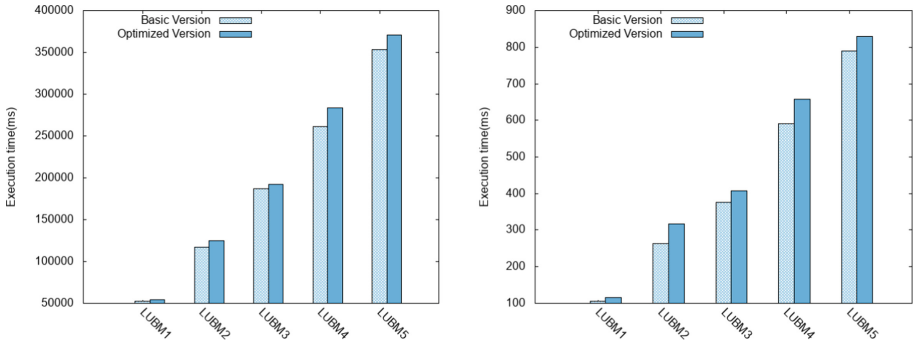
We use an RDF data benchmark LUBM [18] in our experiments. This benchmark is based on an ontology called Univ-Bench for the university domain. Univ-Bench describes universities and departments and the activities that occur at them. The test data of LUBM is synthetically generated instance data over that ontology. Meanwhile, LUBM offers 14 test queries over the data. Table 1 shows the characteristics of the datasets in our experiments.

Table 1. Characteristics of datasets

Dataset	File#	Total size (MB)	V#	E#	Triples#
LUBM1	15	13.6	17,150	23,586	103,397
LUBM2	34	29.3	39,510	54,342	237,210
LUBM3	50	43.5	57,652	79,933	348,105
LUBM4	71	61.1	82,204	113,745	493,844
LUBM5	93	79.6	107,110	148,846	493,844

5.2 Experimental Results

The experimental results show the effectiveness and efficiency of our ontology-aware unified storage scheme. Data insertion and deletion can be completed efficiently. Meanwhile, our scheme is more compact than Neo4j and the traversal time of the triples in our scheme is less than that in Neo4j on all data sets. In our experiments, we provide two versions of the unified storage schemes, one of which is a basic version without the ontology information, while the other is an optimized version with the ontology information.



(a) Data Insertion (b) Data Deletion

Fig. 7. The experimental results of data insertion and deletion on LUBM datasets.

Data Insertion and Deletion. The experimental results show that the average execution speed of the basic version is 4.73% and 9.52% faster than that of the optimized version in data insertion and deletion, respectively, which is not counterintuitive. Though the optimized version is slightly inferior to the basic version, it is complete with the respect to the semantic information. Therefore, it is quite important to measure the tradeoffs between efficiency and reasoning capability.

Actually, the time costs of two versions are on the same order of magnitude, thus are comparative, as shown in Fig. 7. Although the performance of the optimized version is not better than that of the basic version, it is worthwhile to trade the slight overhead for reasoning capability.

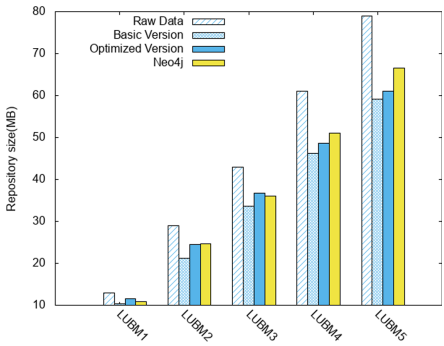


Fig. 8. The experimental results of repository size while inserting data.

Repository Size. We check the change of the file size before and after the data sets being loaded. Experiments were conducted to compare the space required

for the optimized version with the basic version and Neo4j, which supports an RDF plugin for importing RDF data.

Compared with the optimized version, the repository size of the basic version is reduced from 5.11% to 13.36%, as shown in Fig. 8. It is reasonable that the optimized version requires more storage space for the hierarchical structure of ontologies. Though the basic version performs better, the repository size of two versions are on the same order of magnitude. Therefore, it is worthwhile to trade a little space for capability. Besides, it can be observed that the optimized version outperforms Neo4j on several data sets, i.e., LUBM2, LUBM4, and LUBM5. The growth rate of repository size in Neo4j is higher than that of our basic and optimized version.

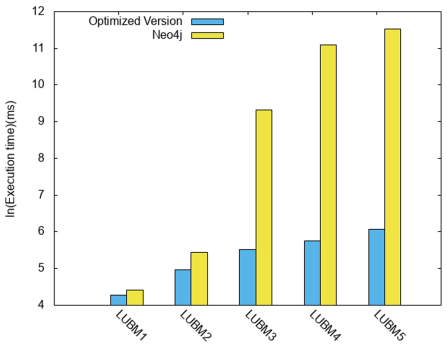
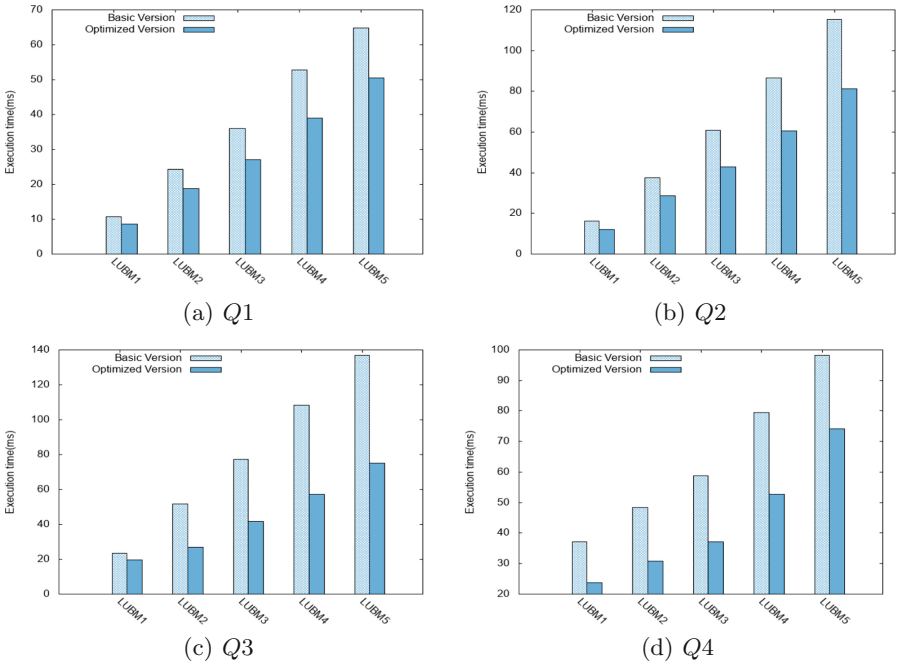


Fig. 9. The experimental results of triple traversal.

Triple Traversal. We also evaluate the time required to traverse all triples in a graph and the experimental results are shown in Fig. 9. The average execution speed in the optimized version is about 99 times faster than that in Neo4j, i.e., our optimization method on average outperforms Neo4j by two orders of magnitude over LUBM data sets. In order to show the results clearly, we change the scale of the Y-axis to logarithmic.

Query Speed. We select four type-related queries from the 14 test queries that LUBM offers: $Q1$ directly refers to the type information of *UndergraduateStudent*, $Q2$ refers to the type information of *GraduateStudent* with some filtering conditions, $Q3$ assumes the *subClassOf* relationship between *Professor* and its subclasses. It is obvious that class *Professor* has a wide hierarchy. $Q4$ assumes the *subClassOf* relationship between *Person* and its subclasses. Similarly, class *Person* features a deep and wide hierarchy.

We compare the basic storage scheme with the optimized storage scheme, and the execution time results of $Q1$, $Q2$, $Q3$, and $Q4$ are shown in Fig. 10. When changing the size of datasets from LUBM1 to LUBM5, query times of these



(a) $Q1$ (b) $Q2$

(c) $Q3$ (d) $Q4$

Fig. 10. The experimental results of efficiency on LUBM datasets.

two schemes have increased. Meanwhile, it can be observed that the optimized version has better query efficiency on all 4 queries. Compared with the basic version, the average query time of the optimized version is reduced from 23.24% to 40.33%.

6 Conclusion

This paper proposes a unified storage scheme for both RDF and property graphs and introduces the concept of ontology to reflect the hierarchical relationships between RDF classes. A prototype system of our storage scheme is designed and implemented based on AgensGraph. Extensive experiments on the LUBM benchmark datasets verify the effectiveness and efficiency of our storage scheme.

Acknowledgments. This work is supported by the National Natural Science Foundation of China (61972275), the Natural Science Foundation of Tianjin (17JCYBJC15400), and CCF-Huawei Database Innovation Research Plan.

References

1. Duan, W., Chiang, Y.Y.: Building knowledge graph from public data for predictive analysis: a case study on predicting technology future in space and time. In: Proceedings of the 5th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, BigSpatial 2016, pp. 7–13 (2016)
2. Wang, H., Fang, Z., Zhang, L., Pan, J.Z., Ruan, T.: Effective online knowledge graph fusion. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9366, pp. 286–302. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25007-6_17
3. Fu, X., Ren, X., Mengshoel, O., Wu, X.: Stochastic optimization for market return prediction using financial knowledge graph. In: 2018 IEEE International Conference on Big Knowledge, pp. 25–32 (2018)
4. Li, Y.: Research and analysis of semantic search technology based on knowledge graph. In: 2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC), vol. 1, pp. 887–890 (2017)
5. Lehmann, J., et al.: DBpedia-a large-scale, multilingual knowledge base extracted from wikipedia. Semant. Web **6**(2), 167–195 (2015)
6. The Neo4j Team: The neo4j manual v3.4 (2018). https://neo4j.com/docs/developermanual/current/
7. TigerGraph Inc.: Tigergraph: the world's fastest and most scalable graph platform (2012). https://www.tigergraph.com/
8. OrientDB Ltd.: Orientdb: first multi-model database (2010). http://orientdb.com/
9. W3C: RDF 1.1 concepts and abstract syntax (2014)
10. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., Vrgoč, D.: Foundations of modern query languages for graph databases. ACM Comput. Surv. **50**(5), 1–40 (2017)
11. Harris, S., Gibbins, N.: 3store: efficient bulk RDF storage. In: PSSS1 - Practical and Scalable Semantic Systems, Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, vol. 89 (2003)
12. Pan, Z., Heflin, J.: DLDB: extending relational databases to support semantic web queries. In: PSSS1 - Practical and Scalable Semantic Systems, Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, vol. 89 (2003)
13. Wilkinson, K.: Jena property table implementation. In: In SSWS, Athens, Georgia, USA, pp. 35–46 (2006)
14. Abadi, D., Marcus, A., Madden, S., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: VLDB, pp. 411–422 (2007)
15. Abadi, D., Marcus, A., Madden, S., Hollenbach, K.: SW-store: a vertically partitioned DBMS for semantic web data management. VLDB J. **18**(2), 385–406 (2009). https://doi.org/10.1007/s00778-008-0125-y
16. Neumann, T., Weikum, G.: RDF3X: a RISC-style engine for RDF. Proc. VLDB Endow. - PVLDB **1**, 647–659 (2008)
17. Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple indexing for semantic web data management. PVLDB **1**, 1008–1019 (2008)
18. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for owl knowledge base systems. Web Semant. Sci. Serv. Agents World Wide Web **3**(2–3), 158–182 (2005)
19. Bitnine-OSS: Agensgraph: a transaction graph database based on PostgreSQL (2017). http://www.agensgraph.org