International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2021, 11(1): 95−120, doi: 10.21655/ijsi.1673-7288.00243
©2021 by Institute of Software, Chinese Academy of Sciences. All rights reserved.

Research Article

KGDB: Knowledge Graph Database System with Unified Model and Query Language

Baozhu Liu (刘宝珠) [1,2], Xin Wang (王鑫) [1,2], Pengkai Liu (柳鹏凯) [1,2], Sizhuo Li (李思卓) [1,2], Xiaowang Zhang (张小旺) [1,2], Yajun Yang (杨雅君) [1,2]

[1] (College of Intelligence and Computing, Tianjin University, Tianjin 300350, China)
[2] (China Tianjin Key Laboratory of Cognitive Computing and Application, Tianjin 300350, China)
Corresponding author: Xin Wang, wangx@tju.edu.cn

Abstract Knowledge graph is an important cornerstone of artificial intelligence, which currently has two main data models: RDF graph and property graph. There are several query languages on these two data models. The query language on RDF graph is SPARQL, and the query language on property graph is mainly Cypher. Over the last decade, various communities have developed different data management methods for RDF graphs and property graphs. Inconsistent data models and query languages hinder the wider application of knowledge graphs. KGDB is a knowledge graph database system with unified data model and query language. (1) Based on the relational model, a unified storage scheme is proposed, which supports the efficient storage of RDF graphs and property graphs, and meets the requirement of knowledge graph data storage and query load. (2) Using the clustering method based on characteristic sets, KGDB can handle the issue of untyped triple storage. (3) It realizes the interoperability of SPARQL and Cypher, which are two different knowledge graph query languages, and enables them to operate on the same knowledge graph. The extensive experiments on real-world datasets and synthetic datasets are carried out. The experimental results show that, compared with the existing knowledge graph database management systems, KGDB can not only provide more efficient storage management, but also has higher query efficiency. KGDB saves 30% of the storage space on average compared with gStore and Neo4j. The experimental results on basic graph pattern matching query show that, for the real-world dataset, the query efficiency of KGDB is generally higher than that of gStore and Neo4j, and can be improved by at most two orders of magnitude.

Keywords knowledge graph; SPARQL; Cypher; RDF graph; property graph

Citation Liu BZ, Wang X, Liu KP, Li SZ, Zhang XW, Yang YJ. KGDB: Knowledge graph database system with unified model and query language. *International Journal of Software and Informatics*, 2021, 11(1): 95−120. http://www.ijsi. org/1673-7288/00243.htm

---

This is the English version of the Chinese article "KGDB: 统一模型和语言的知识图谱数据库管理系统. 软件学报, 2021, 32(3): 781−804. doi: 10.13328/j.cnki.jos.006181".
Foundation items: National Key R&D Program of China (2019YFE0198600); National Natural Science Foundation of China (61972275); CCF-Huawei Database Innovation Research Plan (CCF-Huawei DBIR2019004B)
Received 2020-07-20; Revised 2020-09-02; Accepted 2020-11-06; IJSI published online 2021-03-31

As an important cornerstone of artificial intelligence (AI), knowledge graphs play a role in the transition of new-generation AI from perception to cognition [1]. There are two major data models of knowledge graphs: resource description framework (RDF) graphs and property graphs. RDF is a standard recommended by World Wide Web Consortium (W3C) for expressing knowledge graphs, which has been widely adopted by triple databases represented by gStore [2]. Property graphs are serving as the underlying data models of graph databases such as Neo4j [3], Dgraph [4], and HugeGraph [5].

The development of relational databases over decades of years shows that unified data model and query language is the key to the advancement in data management technology. With regard to the inconsistent data models, storage schemes, and query languages in the management of knowledge graph databases, we develop a knowledge graph database (KGDB) system with a unified data model and query language.

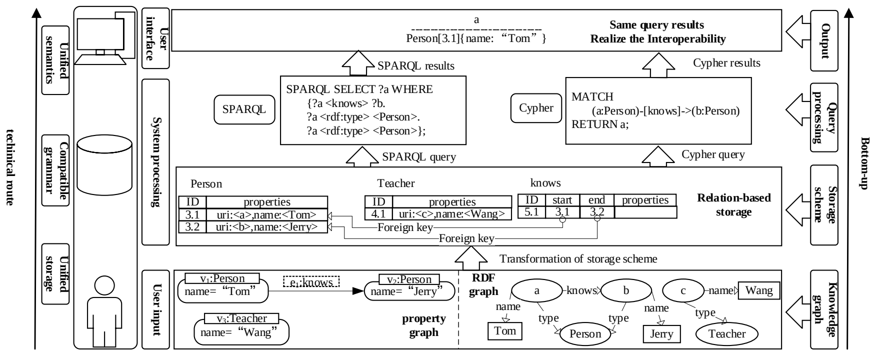
The unified storage scheme allows KGDB to store RDF graphs and property graphs in partitions according to the types of entities. Characteristic set (CS)-based clustering is used to classify untypedentities into ones that are semantically similar. Query interfaces of RDF graphs and property graphs are provided for SPARQL and Cypher to operate on the same knowledge graph, realizing their interoperability. KGDB follows a technical route of “unified storage–compatible grammar–unified semantics.” In underlying storage, a unified storage scheme is used for processing two data models of knowledge graphs; in query expression, two grammatically different query languages meant for different data models of knowledge graphs are compatible; in addition, the grammatically different query languages are aligned into the unified semantics, enabling the same query processing method.

The overall architecture of KGDB is shown in Figure 1, in a bottom-up way.

(1) In the user input layer, data about RDF graphs and property graphs can be input;

(2) In system processing, there are two steps. First, data can be transformed into relational tables where they are clustered by types in terms of the unified storage scheme, based on which the raw knowledge graph data are stored. Second, two query languages are allowed to operate on the same knowledge graph in query processing;

(3) In the user interface layer, the basic graph pattern matching query results can be seen in standard format.



```
SPARQL SELECT ?a WHERE
    {?a <knows> ?b.
    ?a <rdf:type> <Person>.
    ?a <rdf:type> <Person>};
```

```
MATCH
    (a:Person)-[knows]->(b:Person)
RETURN a;
```

ID	properties
3.1	uri:<a>,name:<Tom>
3.2	uri:<b>,name:<Jerry>

ID	properties
4.1	uri:<c>,name:<Wang>

ID	start	end	properties
5.1	3.1	3.2	

Figure 1. Architecture of KGDB

Given the diverse data models of existing knowledge graph database management systems, different storage schemes are proposed. On the one hand, there are three categories of schemes for the storage of RDF graphs. First, they can be stored by triples such as triple tables and horizontal tables. Second, they can be stored by their data types, including property tables, vertical partitioning[6], sextuple indexing, and DB2RDF[7]. Third, they can be classified and stored by semantic information, such as characteristic sets (CSs)[8], extended characteristic sets (ECSs)[9], and R-type[10]. On the other hand, native schemes are often adopted for property graph storage, including Neo4j, JanusGraph[11], and TigerGraph[12].

Fundamentally, knowledge graph data are different from relational data because they are flexible, which poses new challenges for conventional storage and query processing. A CS-based clustering method is proposed, so that entities are clustered by types in terms of predicate groups, and untyped entities are also clustered by relational characteristics, realizing the unified storage of knowledge graphs.

Due to different data models of existing knowledge graph database management systems, their query processing is completed with different languages: SPARQL on RDF graphs and Cypher on property graphs. The difference in grammar hinders their interoperability in a unified storage scheme. For this reason, SPARQL and Cypher are semantically aligned to operate on the same knowledge graphs without distinguishing underlying data models.

Extensive experiments on real-world datasets and synthetic datasets reveal that KGDB is more efficient than gStore (RDF graph database) and Neo4j (property graph database) in the storage management and query processing of knowledge graphs.

In this paper, our contributions can be summarized as follows:

(1) We propose a unified storage scheme for knowledge graphs based on relational models that clusters data by types, realizing the efficient storage of RDF graphs and property graphs; the dictionary encoding method is used to compress storage space, catering to the smooth storage and query of knowledge graph data.

(2) We propose CS-based clustering, with which untype entities are also clustered into data with similar predicate groups for their storage; thus, the unified storage scheme can be applied to flexible and variable semi-structured data.

(3) SPARQL and Cypher are compatible in our KGDB, which allows the interoperation through semantic alignment, so they can operate on the same knowledge graph.

(4) Extensive experiments on real-world datasets and synthetic datasets are carried out, revealing that KGDB is validated as it is more efficient than gStore and Neo4jin the storage management and query processing of knowledge graphs.

We introduce the related work in Section 1 and propaedeutics in Section 2, describe the unified storage scheme of RDF graphs and property graphs in KGDB in Section 3, explain the way to the interoperation between query languages in Section 4, show the extensive experimental results in Section 5, and make a conclusion in Section 6.

1 Related Work

Storage schemes and query processing ways are emerging with the advancement in knowledge graphs. In this section, the storage schemes and query processing methods of two existing data models of knowledge graphs will be introduced. The increasing scale of knowledge graph data makes further requirements for storage and query, so distributed knowledge graph database management systems attract wide academic attention [13,14].

1.1 Storage schemes of knowledge graphs

1.1.1 *RDF graph*

(1) Storage directly by RDF triple features

Triple tables store data into a three-column table, represented by 3store [15]. Horizontal tables store all the predicates and objects corresponding to a subject in a row, which are adopted by the DLDB system [16]. Property tables, employed by Jena [17], store subjects of the same type in one table. Vertical partitioning establishes two-column tables for each predicates to store the subjects and objects connected by it, which is applied to SW-Store [18] and TripleBit[19]. Sextuple indexing sacrifices storage space to speed up query processing by storing all the six permutations of triples and create indexes on the first column, used by RDF-3X [20] and Hexastore[21]. Instead of binding a column with a certain predicate, DB2RDF [7] dynamically maps the predicates into columns through hashing and handles multi-value predicates with extra tables, which is systematically adopted by DB2RDF.

Such schemes are intuitive and simple but faced with sparsity and null values. Moreover, a subject may correspond to several predicates, leading to more diverse predicates connected to different subjects than our expectation. Also, subjects of the same type may correspond to so quite different predicates that they cannot be neglected. In these schemes, there can be many null values in the relational tables that are so sparse and seriously affect storage.

(2) Storage by RDF semantics

CS-based storage scheme[8], adopted by the RDF-3X system [20],divides RDF graph data by star patterns. Entities with the same predicate groups are clustered, largely reducing the number of tables. However, this method equally processes all the predicates and may cluster most entities into one type, leading to poor division. ECS-based storage [9] conducts muti-layered division by star patterns and generates second-level indexes, speeding up query, but this approach also divides too many entities into one set. Ontologies are introduced into the R-Type [10]models, employed by SemStorm[22] to divide triples by predicates into inferable and non-inferable ones. Inferable triples in star patterns are not stored physically to save space. Star patterns with inferable triples are mapped directly into right patterns to accelerate the query. Nevertheless, R-Type models cannot divide untyped entities.

Although these methods are more accurate and can optimize storage with semantic information, few relational storage schemes are developed and most of the existing ones have only prototypes, which are far from application. Relational databases are now stable in the research on transaction management and scalability, so more support is available using relational storage schemes.

1.1.2 *Property graph*

(1) Relational storage scheme

SQLGraph[23] stores property graphs based on relation tables and JSON key values in a relational scheme. It hashes every edge label into two-column relation table and stores adjacency lists of the edges in that table: edge labels in one column and corresponding values in the other. AgensGraph[24] is amulti-model graph database based on the relational model. It separately stores vertices and edges of property graphs in relation tables by labels, and records their attribute values in the format of JSON.

Considering the semi-structured data, the relational scheme is not flexible enough for the storage of property graphs. The cost is not acceptable to change the structure of relational tables once they are created. A more flexible and more efficient scheme is needed to store billions of vertex and edge data of knowledge graphs.

(2) Document-based storage scheme

MongoDB [25] is a database system with document-based storage scheme. It provides Web applications with scalable high-performance data storage alternative and stores data as a document.

The data structure is composed of key-value pairs and supports nested objects. In Neo4j, each data record, or node, stores direct pointers to all the nodes it's connected to, does not need extra relational databases or NoSQL databases. It stores property graph data in the native graph database for various query demands.

The document-based storage scheme is often applied in distributed environments which have stricter requirements for data storage than standalone environments. The storage efficiency of this scheme fails to meet the requirements for the storage and query of increasing larger-scale property graphs. Current storage schemes are mostly intended for certain knowledge graphs and are not ready for wider application. For this reason, it is necessary to develop a unified and efficient relational storage scheme for the two mainstream data models of knowledge graphs.

1.2 Query processing of knowledge graphs

1.2.1 *RDF graph*

Blazegraph[26] is a graph database management system based on anRDF triple base, whichapplys to RDF triples and SPARQL queries. Jena [17] is the open-source framework and RDF triple based in semantic Web. It follows the W3C standard, supports the query processing with SPARQL, and includes a set of rule-based inference engines for RDFS and OWL ontology inference tasks. gStore [2] uses signature graphs corresponding to RDF graphs, creates VS-tree indexes, and allows the query processing with SPARQL. Virtuso[27] is a hybrid database management system that well supports the Linked Data protocols of W3C. RDF4J [28], evolving from Seasame framework developed by Aduna, supports SPARQL 1.1 and allows the analysis, storage, inference, and query of RDF data. RDF-3X [29]specifically establishes a compression storage scheme and a technique for the query processing and query optimization of RDF data. The AllegroGraph system [30] completely supports semantic inference. GraphDB[31] includes the SAIL layer of the RDF4J framework and supports RDF inference through the built-in rule-based "forward-chaining" inference engine.

1.2.2 *Property graph data*

Neo4j [3], a property graph database system, is the most widely adopted graph database and supports Cypher. AgensGraph[24] stores property graphs based on relational models and establishes the Cypher processing layer based on PostgreSQL. JanusGraph[11] is an open-source distributed graph database with separated storage backend and query engine. It includes a MapReduce-based graph analysis engine and can transform Gremlin navigation queries into MapReduce tasks. OirentDB[33] is designed for the storage of graph and document data and supports SQL and Gremlin navigation queries for graphs and the MATCH statement similar to Cypher. Cypher for Apache Spark[34] offers Cypher engine based on Spark framework.

The two data models now have their respective query languages, grammar, and semantics. Although several optimization methods are proposed on specific systems, they still hinder the proliferation of knowledge graph query. Therefore, it is essential to develop a semantically interoprable system that supports both SPARQL and Cypher.

2 Preliminary Knowledge

In this section, the detailed background knowledge will be introduced, including the definitions of RDF graphs and property graphs. Table 1 lists major notations and their meanings.

Table 1 List of symbols

Symbol	Meaning	Symbol	Meaning
$G = (V, E, \Sigma)$	RDF graph G	$\varphi(t)$	Triple classification
$G = (V, E, \eta, src, tgt, \lambda, \gamma)$	Property graph G	$I_C(s)$	CS of entities
$t = (s, p, o)$	Triple	$hist(C)$	Statistics of entity clusters
$\alpha = (a, Lab, Map)$	Property graph vertex pattern	$DCS_{\text{cluster}}(C_j, C_k)$	Distance from C_j to C_k
$\beta = (d, Lab, a, Map)$	Property graph edge pattern	$\lambda{:}(V \cup E)\rightarrow\mathcal{L}$	Mapping from vertexes and edges of property graphs onto labels
T	Finite set of triples	$\gamma{:}(V \cup E) \times \mathcal{K}\rightarrow Val$	Mapping from property onto values
μ	Matching	$lab{:}E\rightarrow\Sigma$	Acquisition of edge labels from RDF graphs

Definition 1 (RDF graph). Let U be the finite set of uniform resource identifiers, L be the finite set of liberals, B be the finite set of blank nodes, and we have $t=(s,p,o)\in U\times U\times(U\cup L\cup B)$ that is the RDF triple (neglecting the blank nodes). $t=(s,p,o)$ means that resource s has a relation p with resource o, namely that resource s has a property p value of o where s, p, and o are subject, predicate, and object, respectively. RDF graph G is the finite set of t. V,E, and Σdenote sets of vertexes, edges, and labels, respectively, with a formalized definition of $V=\{s|(s,p,o)\in G\}\cup\{o|(s,p,o)\in G\}$,$E\subseteq V\times V$ and $\Sigma=\{p|(s,p,o)\in G\}$. Function $lab{:}E\rightarrow\Sigma$ returns edge labels in G.

Example 1: As shown in Figure 2, the RDF graph depicts a musical knowledge graph containing such resources as Beethoven (Composer), Lang Lang (Pianist), and Fate Symphony (Music) with several attributes of the entities and relationships such as composes and plays. Ellipsesre present resources, while rectangles stand for literals in the graph. Directed line connecting vertexes show the relationships between vertexes that starts from the subject, passes the edge label, namely predicate, and ends up with the object. The built-in predicate rdf:typerefters to the type of property. For example, the triple (Beethoven,rdf:type,Composer) demonstrates that Beethoven is a composer.

Definition 2 (Property graph). For a property graph $G=(V,E,\eta,src,tgt,\lambda,\gamma)$, V is the finite set of vertexes; $E$is the finite set of edges with $V\cap E=\varnothing$; function $\eta{:}E\rightarrow(V\times V)$ shows the mapping from an edge onto an vertex pair. For example, $\eta(e)=(v_1,v_2)$ shows the directed edge e from v_1 to v_2. Function $src{:}E\rightarrow V$ denotes the mapping from an edge onto the starting vertex($src(e)=v$ shows that edge e starts from vertex v). Function $tgt{:}E\rightarrow V$ means the mapping from edge e onto the end vertex ($tgt(e)=v$ shows that edge e ends up with vertex v). Function $\lambda{:}(V\cup E)\rightarrow\mathcal{L}$ is the mapping from vertexes or edges onto labels (where $\mathcal{L}$ is the set of labels), such as for $v\in V$(or $e\in E$), if$\lambda(v)=l$(or $\lambda(e)=l$), l is the label of vertex v (or edge e). Function $\gamma{:}(V\cup E)\times\mathcal{K}\rightarrow Val$ denotes the associated attributes of vertexes or edges, where $\mathcal{K}$ is the set of attributes and Val is the set of values, such as for $v\in V$(or $e\in E$),$property\in\mathcal{K}$, if$\gamma(v,property)=val$(or$\gamma(e,property)=val$), then the *property* of vertex v (or edge e) is *val*.

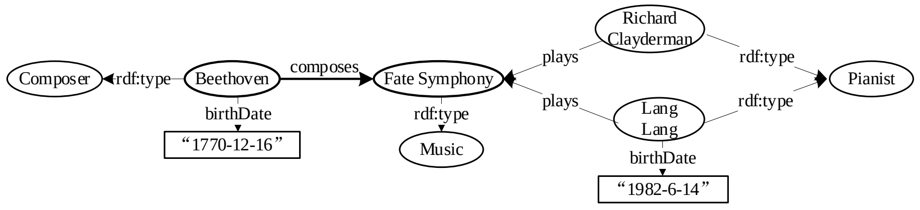


Figure 2. RDF graph

Example 2: Figure 3 shows the property graph of Figure 2, where every vertex and every edge have a unique integer identifier, namely id (such as v_1 and e_1). Every vertex and every edge can be given a label (v_1 has a label Composer) and a attribute composed of attribute key and attribute value (for example v_1 has an attribute that name="Beethoven").

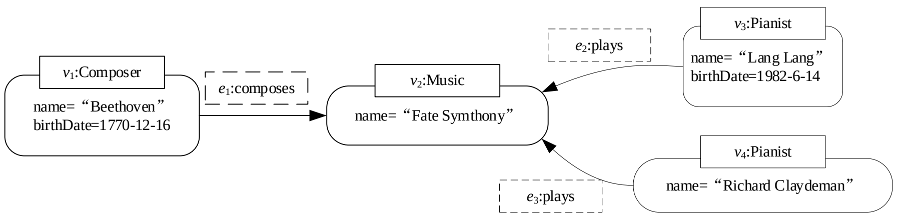


Figure 3. Property graph

3 Storage Scheme for Knowledge Graphs

In this section, we will explain our unified storage scheme for knowledge graphs, which can uninterruptedly accommodate both RDF graphs and property graphs based on the relational models. Meanwhile, the Characteristic sets-based clustering algorithm is adopted to process the untyped data for better supporting the storage of knowledge graph data.

3.1 Storage models for knowledge graphs

The unified scheme stores each entity into its corresponding vertex relation tables v_n ($n\in[1,i]$) and stores each relation into its corresponding edge relation tables e_m($m\in[1,j]$), where i and j show the number of vertex and edge types. Relation tables are named after the types of entities or relations. The relation vertex relations consist of two columns, that is, id (primary key) and property (including attribute and the corresponding value of the attribute); the relation edge relations consist of four columns, that is, id (primary key), start (start vertex of edges), end (end vertex of edges), and property of edge (including attribute and the corresponding value of the attribute). Vertex and edge relations tables are further divided according to the types of entities and relations, avoiding the poor accessibility caused by the excessive data in a single relation table.

3.1.1 *Mapping from the RDF graph to unified storage model*

There are three types of triples in RDF graph data, which are defined below.

Definition 3 (Triple classification). Let C be the set of classes of triples. $C=\{mem,prop,edge\}$ denotes the types of triples, namely type member triples, property description triples, and edge triples. Function $\varphi:T\rightarrow C$ is the mapping from a triple onto its type.

$$\varphi(t)=\begin{cases} mem, & \text{when } t\in\{t=(s,p,o)\mid(s,p,o)\in T\wedge p=\text{rdf:type}\} \\ prop, & \text{when } t\in\{t=(s,p,o)\mid(s,p,o)\in T\wedge o\in L\} \\ edge, & \text{when } t\in\{t=(s,p,o)\mid(s,p,o)\in T\wedge p\neq\text{rdf:type}\} \end{cases} \quad (1)$$

According to Definition 3, the example RDF graph shown in Fig. 2 describes a music RDF graph where φ((Beethoven,rdf:type,Composer))=*mem*,φ((Beethoven,birthDate,1770-12-16))=*prop*, and φ((Lang Lang,plays,Fate Symphony))=*edge*.

The vertex relations and edge relations of RDF graph should be created in terms of labels of vertice and edges in graph G. Entities and relations can be shreded into vertex relations and edge relations through **Rules** 1–3.

Rule 1. For any t=(s,p,o), if $\varphi(t)$=mem, a record with an id of s is inserted into the vertex relation o.

Rule 2. For any given t=(s,p,o), if $\varphi(t)$=$prop$, p and o are inserted into the column "property" corresponding to entity s as a key-value pair.

Rule 3. For any given t=(s,p,o), if $\varphi(t)$=$edge$, a record starting from s and ending up with o is inserted into edge relation p.

3.1.2 *Mapping from the property graph to unified storage model*

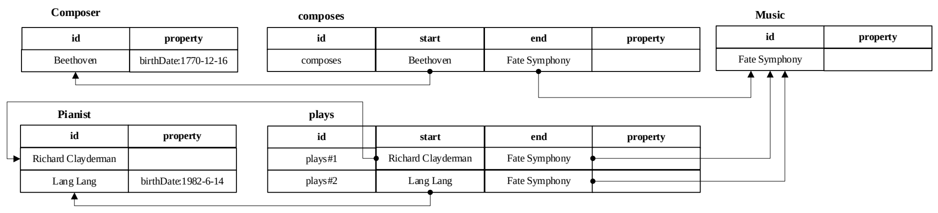
With the built-in support for property of vertexes and edges, property graphs can be easily mapped to the unified storage model depending on **Rules** 4 and 5.

Rule 4. A unique id is assigned to entity v and a record with id is inserted into edge relation $\lambda(v)$($\lambda(v)$ is the label of entity v). The key-value pairs representing the vertex attributes will be inserted into the second column of relation $\lambda(v)$."

Rule 5. A unique id is assigned to relation e according to its edge label $\lambda(e)$ in the property graph and a record with id is inserted into edge relation$\lambda(e)$. The key-value pairs representing the edge attributes will be inserted into the last column of relation $\lambda(e)$; the id of $v_1$will be kept into the column "start"; the id of $v_2$will be kept into the column "end" ($\eta(e)=(v_1,v_2)\wedge src(e)=v_1\wedge tgt(e)=v_2$).

In property graphs, an id in the vertex relation is just an identifier without actual meaning, while that in RDF graphs corresponds to an actual URI (with actual meaning). For unified expression, the URI of v, namely v_{uri}, is added into the second column "property" in the vertex relations as a new attribute, i.e., $\gamma(v,uri)=v_{uri}$.

Example 3: According to the above rules, the examples in Figures 2 and 3 can be transformed into a unified relation-based storage scheme, as shown in Figure 4. Entities are stored into vertex relations in terms of their types (Composer, Pianist, and Music), and the relations can be stored into edge relations in accordance with their types (composes and plays). Arrows demonstrate foreign-key relations.



Composer

id	property
Beethoven	birthDate:1770-12-16

composes

id	start	end	property
composes	Beethoven	Fate Symphony	

Music

id	property
Fate Symphony	

Pianist

id	property
Richard Clayderman	
Lang Lang	birthDate:1982-6-14

plays

id	start	end	property
plays#1	Richard Clayderman	Fate Symphony	
plays#2	Lang Lang	Fate Symphony	

Figure 4. Unified storage scheme for knowledge graphs

Knowledge graph data are stored with this unified storage scheme according to types of entities and relations. Edge property is not included in Figure4, so the column "property" is null in the corresponding table. The type information of entities and relations is supported by labels in property graphs and the built-in keyword rdf:type in RDF graphs. It is reasonable to divide vertices and edges for storage management in terms of their types and reduce the data redundancy and sparsity of existing schemes.

After the establishment of relational tables, functions of relational tables will be given to every operation with the table names as objects. The relational table set of storage scheme is $R=\{r_1,r_2,\dots,r_n\}$, and its corresponding name set is $X=\{x_1,x_2,\dots,x_m\}$; Function $name$:$R\to X$ returns the name of a relational table; Function rel:$T\to R$ returns the relational table of an entity t, where T is the set of entities and $t\in T$.

More than one relation may exist between two entities, namely multiple properties. With the storage scheme mentioned in Section 3.1, KGDB is used to store several attribute key-value pairs for a single subject. Most existing storage schemes map URIs or literals to integer identifier through dictionary encoding. In other words, the mapping technique effectively realizes the conversion from string to id, and reduces the space overhead of database to the minimum. KGDB adopts the dictionary encoding method similar to most existing methods, and compresses the space resources required by the storage schemes.

3.2 Optimum storage scheme for untyped entities

In the unified storage model introduced above, knowledge graph data are divided by vertex and edge types and stored in the corresponding tables of vertexes and edges, neglecting the storage of untyped entities. Basically, all untyped entities are equally stored in a single relation table. Such method may lead to oversized relation tables when dealing with datasets with a large number of untyped entities, reducing the efficiency of queries. Meanwhile, untyped entities are not connected without semantic information, which is opposite to the assumption that semantically equal or similar entities should be stored in close space.

Untyped entities are divided into the closest type by the CS-based clustering algorithm. The hierarchical clustering algorithm can give the similarity between nodes according to a certain distance function and merge the nodes step by step depending on the similarity. When the certain condition is reached, the merging will be terminated. In this section, the CS and CS distance of entities will be defined to measure the similarity between entities to help store untyped entities.

3.2.1 *CS of entities*

In RDF graphs, several triples are used to describe the characteristics of an entity. The CS [8] can be defined as the name set of edges starting from the entity vertex, which will be introduced in detail.

Definition 4 (Characteristic set of entities). The characteristic set I of entity s in the knowledge graph dataset D is

$$I_C(s) = \{p|\exists o:(s, p, o) \in D\} \tag{2}$$

Example 4: In an RDF dataset, there are 3 triples that describe entity s_1, the novel *The Old Man and the Sea*, i.e., (s_1, title, “The Old Man and the Sea”), (s_1, author, Hemingway), and (s_1, year, “1951”). The CS of s_1 is $I_C(s_1)$ = {title, author, year}.

3.2.2 *Entity cluster and distance*

Cluster C contains several entities. To better express the property characteristics of all the entities in a cluster, for any given cluster with some entities, we define *hist*(C) that demonstrates the statistics of CSs of entities in this cluster and records the number of property m of all entities. *hist*(C) can be defined as the set of key-value pairs of every property and its count:

$$hist(C) = \bigcup_{i=1}^{m}(property_i, count_i) \tag{3}$$

where $property_i$ is property i; $count_i$ is the count of property i in cluster C. *hist*(C) can then be used to define the distance between two clusters containing some entities. The distance between C_j and C_k can be expressed by the distance between their statistics of entities:

$$DCS_{\text{cluster}}(C_j, C_k) = DCS_{hist}(hist(C_j), hist(C_k)) = \sum_{p_i \in \{p_1,\ldots,p_n\}}(count(C_j, p_i) + count(C_k, p_i))b_i \tag{4}$$

where

- n is the total number of properties in the two clusters;
- b_i denotes whether property i exists simultaneously in the two clusters, if so, $b_i = 0$; or else $b_i = 1$;

- *count*(C_j,p_i) and *count*(C_k,p_i) are the counts of p_i in C_j and C_k, respectively.

In view of Formula (4), the distance between two clusters are the sum of counts of all the property that does not exist simultaneously in the two clusters. The counts of properties implies the importance to the cluster. For example, the properties of authors and titles in entities with the type of book have high counts. In this way, the similarity between two clusters can be measured.

3.2.3 *CS-based entity clustering algorithm*

The definitions of entity-based CSs, statistics of CSs of clusters containing several entities, and the cluster distance can be used for the optimization of the unified storage scheme. With the hierarchical clustering, an entity cluster algorithm is proposed based on entity types, which can divide a untyped entity into a known type by clustering.

For entity $s \in S$, S is the set of entities; Function *haveType*:$S \rightarrow${TRUE,FALSE} returns whether an entity is typed or not. If s is typed, the return is TRUE; otherwise it is FALSE. Function *getType*:$S \rightarrow TYPE$ returns the type of an entity where $TYPE$ is the set of entity types.

We need to calculate distances between clusters and find the closest ones for merging. For the set of entity clusters $C=\{C_1,C_2,\ldots,C_n\}$, Function *findMin*(C) calculates distances between every two clusters and gives the indices of the closest ones.

Every entity is considered as a cluster, and clusters are merged in a bottom-up manner according to the similarity between CS of entities. The ones with entities totally different in types should not be merged.

Algorithm 1 realizes the CS-based entity clustering, which can divide entities into clusters by their types. The entities in a cluster have similar property, that is, the entities in each cluster tend to have the same type. Algorithm 1 first merges entities of the same type into a cluster, and regards every untyped entity as a single cluster (rows 2–8). Clusters are then merged according to cluster distance DCS_{cluster} in a bottom-up manner. It finds out the closest C_i and C_j($DCS_{\text{cluster}}(C_i,C_j)$ is the lowest) from the known cluster set C and requires that the two are neither typed clusters (rows 10–12). Then, it merges the two clusters, assigns the known type C_i to the merged cluster, and updates *hist*(C_i) (rows 14–15). It repeats merging until there are no clusters to be merged. Clusters with untyped entities are merged into those with typed entities by entity clustering, and entities in a cluster are of the same type.

Algorithm 1. Clustering of typeless entities of triples

Input: Entity set S;
Output: Set of entity clusters C.

```
1.  for each $s \in S$ do
2.    if $haveType(s)$ then
3.      $\tau \leftarrow getType(s)$;          //add entity s to the cluster with type-$\tau$ entities
4.      $C_\tau \leftarrow C_\tau \cup \{s\}$;
5.      $C \leftarrow C \cup \{C_\tau\}$;
6.    else
7.      $C_0 \leftarrow C_0 \cup \{s\}$;        //add every untyped entity as a cluster to $C$
8.      $C \leftarrow C \cup \{C_0\}$;
9.  end
10.  while $|C|>1$ do
11.   $i,j \leftarrow findMin(C)$;            //obtain the indices of the two different closest clusters
12.   if $i=0 \wedge j=0$ then               // find no suitable clusters
13.     break;
```

```
14.   $C_i \leftarrow C_i \cup C_j$;                //merge $C_i$ and $C_j$
15.   $C \leftarrow C \backslash C_j$;
16. **end**
```

Example 5: The clustering process is elaborated.

At the beginning, s_1 and s_2 with rdf:type of books are merged into C_1 where

- $I_C(s_1)$ = {title, author, year};
- $I_C(s_2)$ = {author, year};
- $hist(C_1)$ = {(title,1), (author, 2), (year, 2)}.

s_3 and s_4 with rdf:type of movies are merged into C_2 where

- $I_C(s_3)$ = {title, director, year};
- $I_C(s_4)$ = {director, year};
- $hist(C_2)$ = {(title, 1), (director, 2), (year, 2)}.

Finally, s_5 without rdf:type is taken as C_3:

- $I_C(s_5)$ = {title, director};
- $hist(C_3)$ = {(director, 1)}.

Then, we have $DCS_{\text{center}}(C_1, C_3) = 6$ and $DCS_{\text{center}}(C_2, C_3) = 3$.

C_3 is closer to C_2 than to C_1 in view of entity types, so it is merged into C_2. Specifically, $s_5$isstored in the vertex table of the type "movie" according to our scheme.

Definition 5 (Set of optimal entity clusters). For the set of optimal entity clusters C, (1) all the clusters in C contain typed entities and two clusters have no entity of the same type;(2)the closest distance entities of all the entities in C are contained in the clusters with corresponding entities.

The correctness and complexity of Algorithm 1 are confirmed as below.

Theorem 1. For any given set of entities S, Algorithm 1 can give the set of optimal entity clusters C.

Proof: Algorithm 1 first classifies data in the dataset according to their characteristics. Typed data are classified into the cluster with type-τ entities i.e., C_τ, which is then merged into the set of entity clusters C. If s is untyped, it is classified into a cluster C_0 for processing untyped entities. Thus, (1) in Definition 5 can be satisfied because every entity has been classified into a cluster after the first round of iteration. During every subsequent round of iteration, the distance between every two clusters are calculated, i.e., $DCS_{\text{cluster}}(C_1,C_2)$. If C_i and C_j ($i \neq j$) with the minimum distance can be found out, they are merged; or else the clustering is terminated. Subsequent iteration ensures (2) in Definition 5 and can get the set of optimal entity clusters C. Q.E.D.

The time complexity of Algorithm 1 results from (1) e iterations and (2) comparison between cluster distances. Under the worst circumstance, every entity forms a single cluster, and the complexity of cluster distance is $O(s^2)$. Therefore, the time complexity of Algorithm 1 is $O(es^2)$.

4 Interopration Between Query Languages

SPARQL on RDF graphs are grammatically different from Cypher on property graphs. The query based on the unified storage scheme introduced in Section 3 can be realized by both SPARQL and Cypher on KGDB, achieving the interopration between query languages. In References [35] and [36], formal semantics of RDF and Cypher are given. The KGDB regards

SPARQL and Cypher as two different grammatical views with unified query semantics. Relational models are used as physical models to store both RDF graphs and property graphs. Meanwhile, the two query languages are semantically aligned.

The query of basic graph pattern(BGP) matching is the basic operator for the query processing of knowledge graphs, which is fundamentally subgraph homomorphism or subgraph isomorphism. Subgraph matching serves as the core operator of most existing knowledge graph query languages. Although there are many query algorithms for subgraph matching of graph data, a systematic and effective one for large-scale knowledge graphs is lacking. KGDB, that queries subgraph matching based on SPARQL and Cypher, needs to align the two languages and transforms a query intention into two expressions, assuring the correctness and efficiency of query processing.

In this section, typical operators in relational algebra, including ρ(rename),π(project), σ(select),$\bowtie$(join),$\times$(Cartesian product),$\cap$(intersect), and $\cup$(unite), will be used. The join list $\mathbb{L}$ shows abstract semantics. In $\mathbb{L}=\{r_1,r_2,\ldots,r_n\}$, $r_1,\ldots,r_n$ are the n relation tables in the join list. $r\rightarrow property$ denotes the *property* of all the triples in relation table r.

4.1 SPARQL query processing

First, the formalized definition of BGP matching in RDF graphs is given.

Definition 6 (BGP matching on RDF graphs). The query of BGP matching on RDF graph G, i.e., Q, is semantically defined:

(1) μ is the mapping of vertex in $V(Q)$ onto that in V;

(2) $(G,\mu)\vDash Q$ if and only if any $(s_i,p_i,o_i)\in Q$ satisfies that: i) s_i and o_i can be matched with $\mu(s_i)$ and $\mu(o_i)$, ii) $(\mu(s_i),\mu(o_i))\in E$, and iii) $lab(\mu(s_i),\mu(o_i)) = p_i$;

(3) $\Omega(Q)$ is the set that allows $(G,\mu)\vDash Q$ to satisfy μ, namely the answer set of BGP query G_Q.

Example 6: Figure5 shows the query of BGP matching Q containing triples t_1=(Beethoven, composes,?music) and t_2=(Beethoven,birthDate,“1770-12-16”). The two triples constitute a simple star structure for querying all the works of Beethoven. Q is operated on the RDF graph in Figure2, and the variable ?music can be matched with Fate Symphony and outputted if specified in the result clause.

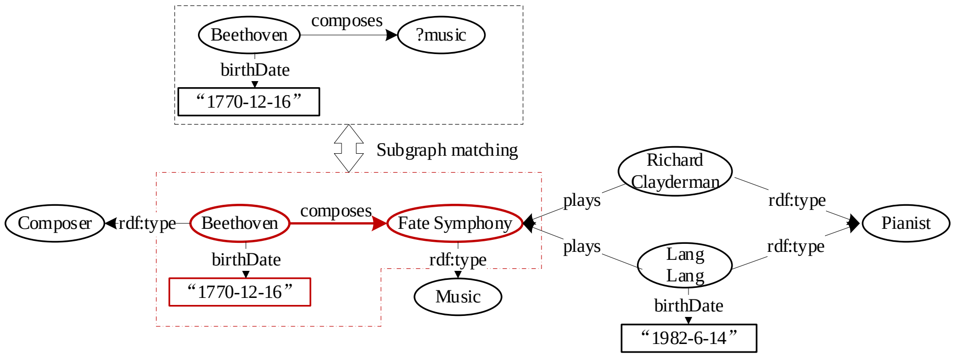


Figure 5. Query of BGP matching

The simplest SPARQL query statement consists of the SELECT clause and the WHERE clause. KGDB semantically analyzes the grammar of SPARQL query statement and generates the semantic tree. In terms of the semantic meaning of every part of SPARQL query statement, a query semantic tree is created from bottom to up in the form of relational algebra, depending on Rules 6–9.

Rule 6. For any triple t=(s,p,o) in the WHERE clause of SPARQL query, if $\varphi(t_i)$=*mem*, a leaf node $r_s=\rho_s(r)$ is added to the bottom layer of the semantic tree where *name*(r)=o. Relation r_s is thus added to the join list$\mathbb{L}$.

Rule 7. For any triple t=(s,p,o) in the WHERE clause of SPARQL query, if $\varphi(t)$=*prop* with $r_s\in\mathbb{L}$, we replace r_s in $\mathbb{L}$ with the intersection between the original $r_s$and the selection operation$\sigma_{r\to p=o}(r)$(where *rel*(s)=r).

Rule 8. For any triple t=(s,p,o) in the WHERE clause of SPARQL query, if $\varphi(t)$=*edge* and $r_s\in\mathbb{L}$, we replacer_s with the one applie] join operation $r_s \bowtie_{r_s.id=r_p.start} r_p$; if $r_o\in\mathbb{L}$, we replace r_o with the one applied join operation $r_o \bowtie_{r_o.id=r_p.end} r_p$; if subject (object) is an URL, a selection operation $\sigma_{r_p.start=s}(r_p)(\sigma_{r_p.end=s}(r_p))$ is applied to the relation table r_p corresponding to the predicate.

Rule 9. For any variable *var* in SELECT clause, a projection operation$\pi_{var.id,var.property}(r_{final})$ is applied, where r_{final} is the final Cartesian product resultof all the relations in join list$\mathbb{L}$.

SPARQL statement can be transformed into query semantic by Algorithm 2.

Algorithm 2. SPARQL query processing of basic graph pattern matching

Input: The set of triples $T=\{t_1,t_2,\cdots,t_{n_1}\}$ where t_i=(s_i,p_i,o_i); the set of variables $VAR=\{var_1,var_2,\cdots,var_{n_2}\}$; the set of relation tables $R=\{r_1,r_2,\cdots,r_{n_3}\}$;

Output: The result r_{result} of the query of BGP matching

```
1.  $\mathbb{L}=\varnothing$;
2.  $r_{result}=\varnothing$;
3.  **for each** $t_i\in T$ **do**
4.  **if** $\varphi(t_i)$=*mem* **then**          //rename relation table$o_i$and add it to the join list
5.  $\mathbb{L}\leftarrow\mathbb{L}\cup\{r_{s_i}\mid r_{s_i}=\rho_{s_i}(r)\wedge r\in R\wedge name(r)=o_i\}$ ;
6.      **elseif** $\varphi(t_i)$=*prop* **then**          //add a selection operation
7.          $\mathbb{L}\leftarrow\mathbb{L}\setminus\{r_{s_i}\}$ ;
8.          $r_{s_i}\leftarrow r_{s_i}\cap\sigma_{r\to p_i=o_i}(r)$, where, $r\in R\wedge rel(s_i)=r$ ;
9.          $\mathbb{L}\leftarrow\mathbb{L}\cup\{r_{s_i}\}$ ;
10.     **else**          //add a join operation
11.         **if** $s_i\in U$ and $_i$is a variable **then**     //thesubject is a constant and the object is a variable
12.             $\mathbb{L}\leftarrow\mathbb{L}\setminus\{r_{o_i}\}$ ;
13.             $r_{o_i}\leftarrow r_{o_i}\bowtie_{.id=r_p.start}(\sigma_{r_p.start=s_i}(r_p))$ ,where $r_p\in R\wedge name(r_p)=p_i$;
14.             $\mathbb{L}\leftarrow\mathbb{L}\cup\{r_{o_i}\}$
15.     **else if** $o_i\in U_i$and $s_i$is a varable **then**     //the object is a constant and the subject is a variable
16. $\mathbb{L}\leftarrow\mathbb{L}\setminus\{r_{s_i}\}$ ;
17. $r_{s_i}\leftarrow r_{s_i}\bowtie_{r_{s_i}.id=r_l}\bowtie_t(\sigma_{r_p.end=o_i}(r_p))$ ,where $r_p\in R\wedge name(r_p)=p_i$;
18. $\mathbb{L}\leftarrow\mathbb{L}\cup\{r_{s_i}\}$
19.     **else**          //the subject and the object are both variables
20. $\mathbb{L}\leftarrow\mathbb{L}\setminus\{r_{s_i},r_{o_i}\}$ ;
21. $r_{o_i}\leftarrow r_{o_i}\bowtie_{r_{o_i}.id=r}\bowtie_t r_p$ ,where $r_p\in R\wedge name(r_p)=p_i$;
22. $r_{s_i}\leftarrow r_{s_i}\bowtie_{r_{s_i}.id=r}\bowtie_t r_p$ ,where $r_p\in R\wedge name(r_p)=p_i$;
23. $\mathbb{L}\leftarrow\mathbb{L}\cup\{r_{s_i},r_{o_i}\}$ ;
24.   **end**
25.   **for each** $var_i\in Var$ **do**      //add a projection operation and output the results in the corresponding relation table
26. $r_{result}\leftarrow r_{result}\cup\pi_{var_i.id,var_i.property}(r_{final})$ ;  //$r_{final}$is the final Cartesian product results of relation tables in $\mathbb{L}$
27.   **end**
```

Algorithm 2 transverses triples involved in SPARQL query and can take different measures to various triples, forming the query semantic expressed by relational algebra. Similar to the storage scheme, the query processing divides t_i=(s_i,p_i,o_i) into three catagories: $\varphi(t_i)$=*mem* that describes the type of an entity, $\varphi(t_i)$=*prop* where the object is literal, and $\varphi(t_i)$=*edge* (the other). For *mem* (line 4 and 5), we add a relation table named after the object o_i to the join list and rename the table as s_i (the other triples in SPARQL query are named after the same variable); for *prop* (line 6–9), we add a selection operation to the set of constraints; for other triples (namely *edge* in line 10–24), we add a join operation. In line 25 and 26, we process all the projection operations and output the final query result according to users' demand.

Theorem 2. The outputs of Algorithm 2 of the set of triples *T* and the set of relation tables *R* in the given SPARQ query are correct.

Proof: Algorithm 2 transverses all the triples involved in SPARQL query and gives a corresponding solution according to the type of $t_i=(s_i,p_i,o_i)\in T$. According to Definition 3, triples have only three semantic types. In other words, Algorithm 2 can work out a corresponding solution to every triple with a join list$\mathbb{L}$ of the right relation table given. All the variables in the SELECT clause exist in the WHERE clause, and the addition of projection operation changes only the final results without affecting the correctness. Q.E.D.

The time complexity of Alogrithm 2 consists of two parts: (1) the algorithm needs to transverse all the triples in the SPARQL query and give corresponding solutions to generate the join list$\mathbb{L}$,which complexity is $O(n)$; (2) the time complexity of adding a new entry to the join list$\mathbb{L}$is a constant $O(k)$.Hence, the time complexity of Algorithm 2 is $O(kn)$.

4.2 Cypher query processing

Similar to the SPARQL query processing method, we first give the formalizeddefinition of property graph pattern matching.

Definition 7 (Property graph pattern). $\alpha=(a,Lab,Map)$is a vertex pattern where $a\in\mathcal{A}\cup\{nil\}$is an optional name; $\mathcal{A}$ is the set of names; *nil* is null; *Lab* is a possibly empty finite set of node labels ($Lab\subseteq\mathcal{L}$); $\mathcal{L}$is the label set of property graphs; *Map* is the possibly empty finite partial map from$\mathcal{K}$to property value *Val*.$\beta=(d,Lab,a,Map)$is an edge pattern where $d\in\{\leftarrow,\rightarrow\}$is its direction; *Lab* is a possibly empty finite label set ($Lab\subseteq\mathcal{L}$);$a\in\mathcal{A}\cup\{nil\}$is an optional name for edge patterns; *Map* is the possibly empty finite partial map from$\mathcal{K}$of properties onto property value *Val*.$\omega=\alpha_1\beta_1\alpha_2\beta_2\ldots\beta_{n□1}\alpha_n$ is a path pattern where α_i is a vertex pattern and β_i is an edge pattern.

Definition 8 (Property graph pattern matching). The recursive definition of property graph pattern matching is as follows[36].

(1) For vertex pattern $\alpha=(a,Lab,Map)$, if the matching is $(v,G,\mu)\vDash\alpha$on property graph *G*, then *a* is *nil* or $\mu(a)=v$,$Lab\subseteq\lambda(v)$and $[[\gamma(v,k)=Map(k)]]_{G,\mu}$;

(2) For path $\mathcal{P}$ with only one vertex, namely *m*=0,if the matching is $(v\cdot\mathcal{P},G,\mu)\vDash\alpha\beta\omega$on property graph *G*, then

   a) *a* is *nil* or $\mu(a)=list(\cdot)$;

   b) $(v,G,\mu)\vDash\alpha$and $(\mathcal{P},G,\mu)\vDash\omega$;

(3) For path $\mathcal{P}$ with $m\geq1$,if the matching is $(v_1e_1v_2\ldots e_mv_{m+1}\cdot\mathcal{P},G,\mu)\vDash\alpha\beta\omega$ on property graph *G*, then

   a) *a* is *nil* or $\mu(a)=list(e_1,\ldots,e_m)$;

   b) $(v_1,G,\mu)\vDash\alpha$and $(\mathcal{P},G,\mu)\vDash\omega$;

   c) $Lab_\beta\subseteq\{\lambda(e_1)\cup\lambda(e_2)\cup\ldots\cup\lambda(e_m)\}$;

   d) $[[\gamma(e_i,k)=Map(k)]]_{G,\mu}$= true;

e) $$(src(e_i),tgt(e_i))=\begin{cases}\{(v_i,v_{i+1})\}, & \text{if } d \text{ is} \rightarrow \\ \{(v_{i+1},v_i)\}, & \text{if } d \text{ is} \rightarrow\end{cases}.$$

The result set of Cypher query *Q* is

$$match(Q,G,\mu)=\biguplus_{\mathcal{P}\text{ in }G}\{\omega\,|\,(\mathcal{P},G,\mu)\vDash Q\} \tag{5}$$

where $\uplus$ denotes bag union.

Similar to SPARQL query statement, the simplest Cypher query includes the MATCH clause and the RETURN clause. In KGDB, Cypher statement is transformed by Rules 10–12.

Rule 10. For all the vertex patterns α=(*a*,*Lab*,*Map*) in Cypher query statements, *n* relation tables, namely $r_1,\ldots,r_n$, are added to the join list$\mathbb{L}$, where $Lab\subseteq\{rel(r_1),rel(r_2),\ldots,rel(r_n)\}$; selection operation $\sigma_{r_i\rightarrow domain(Map)=range(Map)}(r_i), i\in[1,n]$ is added, where *domain*(*Map*) is the domain of *Map* and *range*(*Map*) the range of *Map*.

Rule 11. For all the edge patternsβ=(*d*,*Lab*,*a*,*Map*) in Cypher query statements, join operation is added to relation table in$\mathbb{L}$:

$$\begin{cases} r_{v_{i+1}} \rhd\lhd_{r_{v_{i+1}}} \bowtie_{start}\ r_e \text{ AND } r_{v_i} \rhd\lhd_{r_{v_i}.id=r_e.end} \bowtie\ = \leftarrow \wedge (src(e),tgt(e))=(v_{i+1},v_i)\wedge Lab\subseteq\lambda(e) \\ r_{v_i} \rhd\lhd_{r_{v_i}.i} \bowtie_{art}\ r_e \text{ AND } r_{v_{i+1}} \rhd\lhd_{r_{v_{i+1}}.id=r_e.e} \bowtie\ d = \rightarrow \wedge (src(e),tgt(e))=(v_i,v_{i+1})\wedge Lab\subseteq\lambda(e) \end{cases}.$$

Rule 12. For all the variables *var* in Cypher query statements, projection operation $\pi_{var.id,var.property}(r_{final})$ is added, wherer_{final} is the final Cartesian product result of the relation tables in $\mathbb{L}$.

Compare the two query languages realized in KGDB, Cypher is easier in semantic transformation than SPARQL, since the latter needs to map triples onto relational algebra by the triple classification, while the former can directly identify the semantics by patterns of all the parts.

Theorem 3. Rules 10–12 can be correctly transform Cypher query statements into query semantics in the form of relational algebra.

Proof: In Cypher statements, MATCH clause is transformed by Rules 10 and 11, while RETURN clause is processed by Rule 12. Transformations are carried out for the vertex pattern and edge pattern in MATCH clause by Rules 10 and 11, respectively: (1)for labeled vertex pattern α=(*a*,*Lab*,*Map*), *n* relational tables, namely $r_1,\ldots,r_n$, are added to the join list$\mathbb{L}$,where $Lab\subseteq\{rel(r_1),rel(r_2),\ldots,rel(r_n)\}$, and selection operation $\sigma_{r_i\rightarrow domain(Map)=range(Map)}(r_i), i\in[1,n]$ is added, where *domain*(*Map*) is the domain of *Map* and *range*(*Map*) the range of *Map*; (2) for labeled edge pattern β=(*d*,*Lab*,*a*,*Map*), two join operations are applied. All the variables in Rule 12 should be included in MATCH clause with projection operation $\pi_{var.id,var.property}(r_{final})$ added, where r_{final} is the final Cartesian product result of relation tables in join list $\mathbb{L}$. All the query matching can be solved in this way. Q.E.D.

Example 7: Figure 6 demonstrates a query of property graph pattern matching about all the works of Beethoven. This query can be executed on the property graph of Figure 3 and get the results in the dotted line.

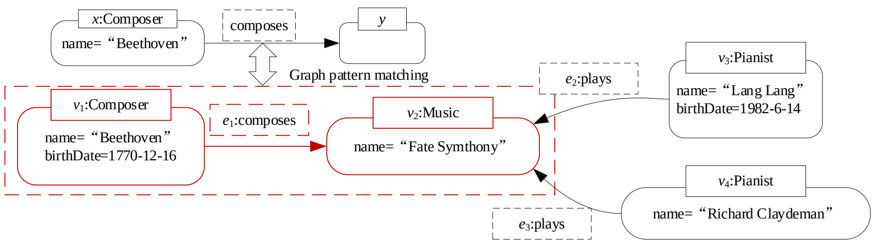
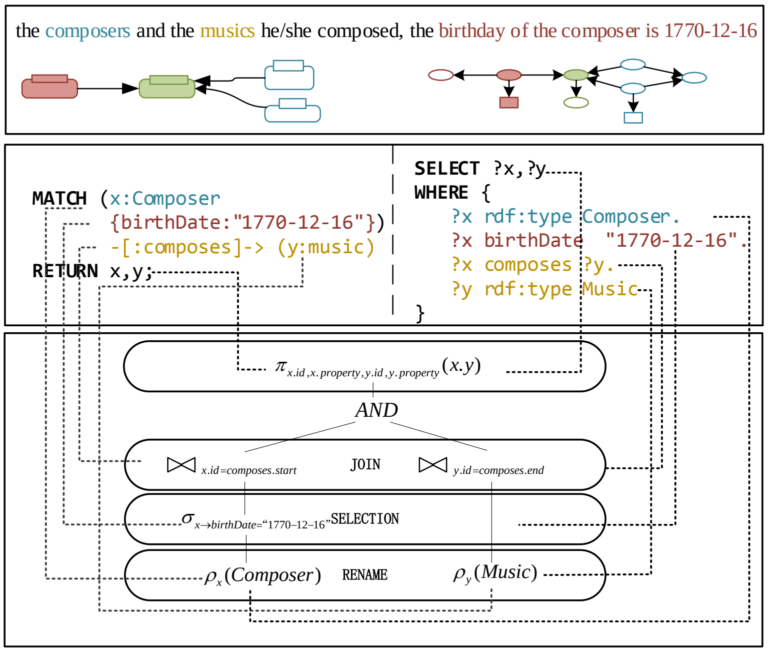


Figure 6. Cypher query of property graph pattern matching

4.3 Semantic alignment

As shown in Figure 7, the semantic alignment between SPARQL and Cypher is carried out by KGDB.As the same semantics can be expressed with totally different grammars, the same query intention can be given in two different languages, which are then transformed into the same query semantics by the respective rules mentioned above. Thus, a unified semantic is available for subsequent query processing.


the composers and the musics he/she composed, the birthday of the composer is 1770-12-16
$\pi_{x.id,x.property,y.id,y.property}(x.y)$
AND
$\bowtie_{x.id=composes.start}$
JOIN
$\bowtie_{y.id=composes.end}$
$\sigma_{x\rightarrow birthDate=\text{“1770–12–16”}}$
SELECTION
$\rho_x(Composer)$
RENAME
$\rho_y(Music)$

```
MATCH (x:Composer
      {birthDate:"1770-12-16"})
      -[:composes]-> (y:music)
RETURN x,y;
```

```
SELECT ?x,?y
WHERE {
    ?x rdf:type Composer.
    ?x birthDate  "1770-12-16".
    ?x composes ?y.
    ?y rdf:type Music
}
```

Figure 7. Semantic alignment

Two query language interfaces are offered on the unified storage scheme as more options for users. The semantic alignment is, in fact, the extension of the two languages.

Example 8: Two transformation processes are shown for better understanding of Figure 7.

(1) SPARQL query

SPARQL query statements are transformed by the rules in Section 4.1:

a) For triples t_1=(?*x*,*rdf*:*type*,*Composer*) and t_4=(?*y*,*rdf*:*type*,*Music*), we have $\varphi(t_1)=\varphi(t_4)=$ *mem* according to Rule 6, so that two relation tables Composer and Music are added to the join list and renamed as$\rho_x(Composer)$ and $\rho_y(Music)$, respectively;

b) For triple t_2=(?*x*,*birthDate*,“1770-12-16”),we have $\varphi(t_2)$=*prop* according to Rule 7.Then we can add selection operation$\sigma_{x\rightarrow birthDate=\text{“1770-12-16”}}(x)$ to renamed table *x*;

c) For triple t_3=(?*x*,*composes*,?*y*),we have $\varphi(t_3)$=*edge* according to Rule 8, thus, a join operation between relation tables is added:

$$r_x \bowtie_{r_x.id=r_{comp}} \bowtie_{start} r_{composes}, r_y \bowtie_{r_y.id=r_{composes}} \bowtie r_{composes};$$

d) For all the variables i.e., *x* and *y* in the RETURN clauses, we add projection operation $\pi_{var.id,var.property}(r_{final})$ wherer_{final} is the final Cartesian product result of relation tables in $\mathbb{L}$, according to Rule 9.

At this point, the SPARQL statement can be transformed into the semantic tree in Figure7.

(2) Cypherquery

Cypher query statements are transformed by rules in Section 4.2:

a) For vertex pattern α=({x,y},{*Composer*,*Music*},*birthDate*→“1770-12-16”),according to Rule 10, we can add relation tables Composer and Music to the join list, rename them as $\rho_x(Composer)$ and $\rho_y(Music)$, respectively, and add selection operation $\sigma_{x\to birthDate=\text{“1770-12-16”}}(x)$;

b) For edge pattern β=(→,{*composes*},*nil*,{·}), according to Rule 11, we can add join operation

$$r_x \bowtie_{r_x.id=r_{composes}.start} r_{composes}, r_y \bowtie_{r_y.id=r_{composes}.end} r_{composes};$$

c) For all the variables i.e., x and y in RETURN clauses, we add projection operation $\pi_{var.id,var.property}(r_{final})$ wherer_{final} is the final Cartesian product result of relation tablesin $\mathbb{L}$, according to Rule 12.

According to the rules in Section 4, the SPARQL and Cypher statements can be transformed into the same abstract semantic tree expressed by relational algebra (alignment methods can be validated according to Theorems 2 and 3, respectively), leading to the unified query semantics for KGDB compatible with two grammatically different query languages. This facilitates subsequent optimization and provides users with another query language option.

5 Experiment

Extensive experiments are carried out on real-world and synthetic datasets to verify the high efficiency of the unified storage scheme and the interoperability between two languages. Moreover, KGDB is compared with gStore [2] and Neo4j [3], which will be explained in detail in this section.

5.1 Experimental settings and datasets

The experiment is deployed on a single-node server. The server has an 8-core Intel(R) Xeon(R) Platinum 8255C@ 2.5GHz CPU, with 32GB of memory, running 64-bit CentOS 7.6 operating system.

KGDB is implemented on the top of AgensGraph, an open-source graph database. Neo4j-community-4.1.0 and neosemantics-4.0.0.1 are used to store RDF graphs in Neo4j. The version of gStore used in the experimentsisgStore-0.7.2. The storage efficiency of KGDB, gStore, and Neo4j is compared. Furthermore, the query efficiency of KGDB on SPARQL BGP matching queries is compared with gStore, and that of Cypher graph pattern matching queries is compared with Neo4j.

LUBM [37], a synthetic dataset, and DBpedia[38], a real-world dataset, are used in this paper. The size of LUBM can be defined by users. 5 LUBM datasets of various sizes are adopted, i.e., LUBM10, LUBM20, LUBM30, LUBM40, and LUBM50. DBpedia is a real-world dataset generated by the extracted entity relationships from Wikipedia. All the datasets are expressed by *N*-Triple. The statistical information of the datasets is shown in Table 2.

Table 2 Datasets

Dataset	Number of triples	Number of vertexes	Number of edges	Document size
LUBM10	1316 700	207 429	630 757	208 M
LUBM20	2782 126	437 558	1332 030	441 M
LUBM30	4109 002	645 957	1967 309	651 M
LUBM40	5495 742	864 225	2630 657	871 M
LUBM50	6890 640	1082 821	3298 814	1.1 G
DBpedia	23 445 441	2257 499	6876 041	3.1 G

8 of the 14 standard queries (Q1–Q6, Q11, and Q14) provided by the LUBM are adopted in the experiments, where

- Q1–Q3and Q14are SPARQL queries without inference:(1)Q1bears large input and high selectivity; (2)Q2is a triangular query involving three entities;(3)Q3contains wide hierarchy class;(4)Q14includes large amount of input data and with low selectivity;
- Q4–Q6 and Q11 involve inference.

gStore does not support inference on RDF graphs and Neo4j only allows inference through a plug-in. Similarly, KGDB does not enable inference query. Hence, we only compare the inference query efficiency of gStore and KGDB that return null value. The inference queries of LUBMs can be divided into four types: (1) Q4–Q9 involve subClassOf relationship; (2) Q5 includes subPropertyOf relationship and cannot be executed without ontology information; (3) Q6–Q10 includes explicit subClassOf relationship, i.e., the hierarchical relationships of entity types involved in query are not directly given in the ontology information; (4) Q11–Q13 need more complex inference, namely those relationship in addition to subClassOf and subPropertyOf should be considered. Every kind of queries is chosen one for the comparison. For a lack of benchmark queries on DBpedia, we design four queries with different data sizes, i.e., Q_dbp1–Q_dbp4. Q_dbp2–Q_dbp4 are structurally equal, of which Q_dbp4 retrieves the greatest amount of results (millions of results), while Q_dbp2 retrieves the lowest amount.

5.2 Results

In this section, a thorough experimental study is conducted to evaluate storage efficiency and query efficiency of KGDB. Every query is issued three times to get the average result.

5.2.1 *Storage time*

As shown in Figure8(a), KGDB spends the least storage time, followed by Neo4j, and gStore the most. With the increase in data size, KGDB can be 10 times faster than gStore and Neo4j in import time and improve the storage efficiency by an order of magnitude than gStore and Neo4j.

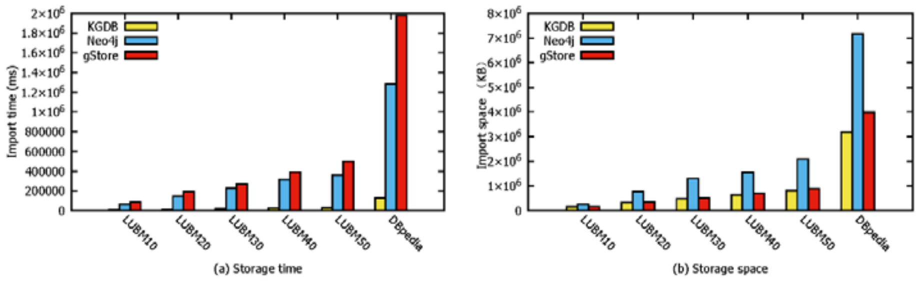


Figure 8. Experimental results of storage time and space

KGDB needs to cluster the untyped entities for once so that it can store datasets for many times without complex transformation. Compared with KGDB, gStore has to transform character strings into id and create VS-trees, whileNeo4j calls for transformation from types into labels.

5.2.2 *Storage space*

As shown in Figure8(b), KGDB outperforms gStore and Neo4j in storage space. Neo4j needs a storage space larger than, and even two times the size of dataset. gStore can compress datasets at a rate of up to 0.8 for storage. Compared with gStore and Neo4j, the compression rate of KGDB can reach 0.7, achieving efficient data storage. With the size of data increasing, KGDB prevails even greater in storage space, due to the utilizing of dictionary encoding. Users only can build only two databases in Neo4j-community, with a graph for each, so that a full backup of Neo4jis required when creating more than one knowledge graphs. Although Neo4j-enterprise supports multiple databases, Neo4j-community provides users with limited services and calls for higher storage space for multiple independent knowledge graphs.

In the experiments, we merely calculate and compare the storage space before and after knowledge graph on loading. The space needed by Neo4j will be larger if the system space is considered.

5.2.3 *Query efficiency*

The experiments are conducted on LUBMs and DBpedia to verify the query efficiency of KGDB. There are four basic queries and four inference queries on LUBM datasets as well as four queries on DBpedia. Q_dbp1 contains large amount of input data and is highly selectable.Q_dbp2–Q_dbp4 are structurally equal, while different in data size. SPARQL and Cypher query statements are made with the same semantics and tested on the three systems.

(1) SPARQL query

gStore is an RDF graph database system and supports SPARQL query, enabling data management of large-scale knowledge graphs. As shown in Fig. 9,gStore fails to support the most complex query Q2, which can be completed by KGDB in a high efficiency. With the data size increasing, the query time taken by KGDB increases in a lower rate than that by gStore for Q3, so KGDB is more efficient than gStore in large-scale knowledge graph query. For Q1 and Q14, the query efficiencies are of the same order of magnitude, so they are comparable.

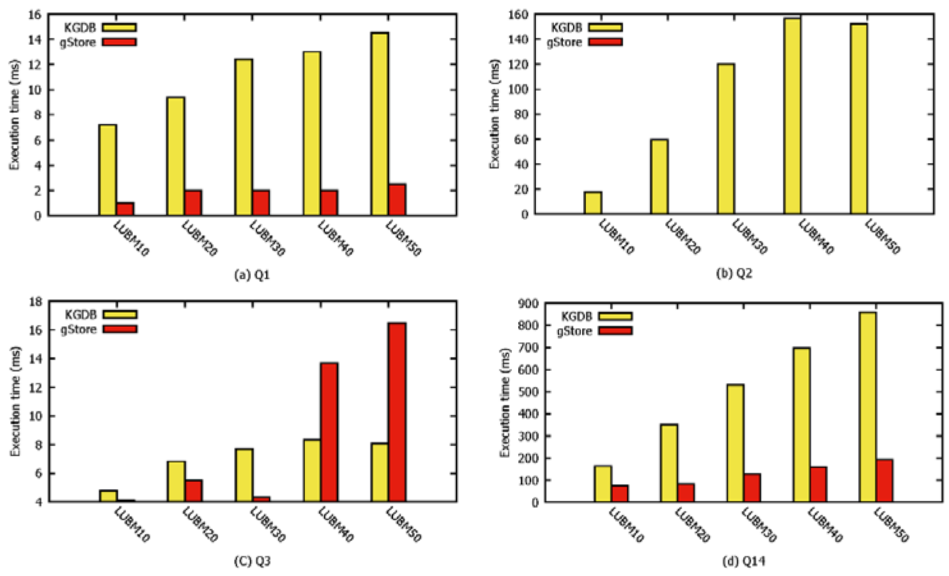


Figure 9. Experimental results of SPARQL queries of KGDB and gStore on LUBMs

The standard benchmark query Q2 on LUBM can lead to system error of gStore, so it cannot be directly executed. To be fair, we do not rewrite the query Q2. Q1 and Q3 are systematically consistent, however, Q3 has a larger size of data. There is a gradual increase in the execution time of Q3, showing the capability of KGDB on large-scale knowledge graph queries. The scale of data volume will not affect query efficiency. For the most time-consuming query Q14,where hundreds of thousands of data will be included in the final result on LUBM50, the query time of KGDB is of the same order of maginitude as that of gStore.

The experimental results of SPARQL inference query efficiency of KGDB and gStore on LUBM datasets are presented in Figure10. The queries can be evaluated in high efficiency in KGDB and gStore, though the results are null because the two systems do not support inference query. For benchmark queries on LUBM datasets, KGDB can complete the queries faster and its query time increases at a lower rate than that of gStore.

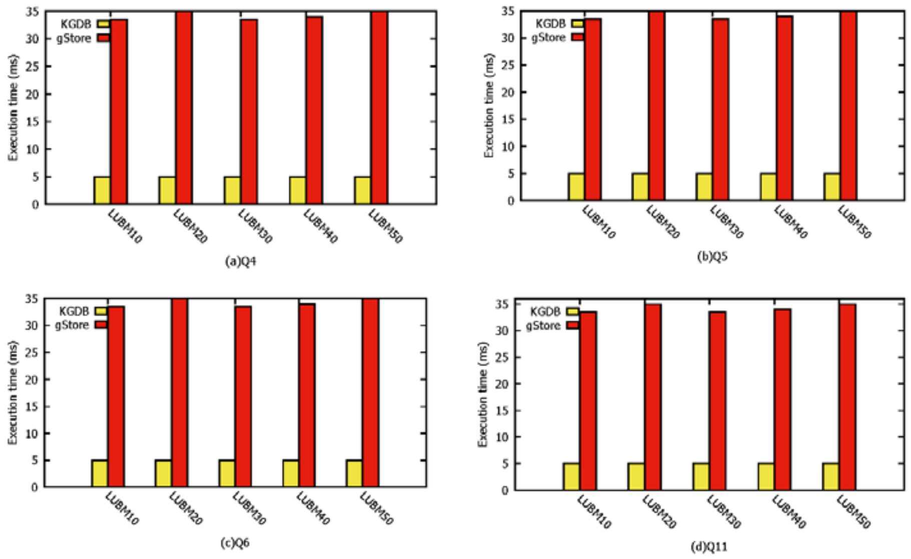


Figure 10. Experimental results of SPARQL Inference query efficiency of KGDB and gStore on LUBMs

According to Figure11, on DBpedia, KGDB spends shorter time than gStore for Q_dbp1–Q_dbp3. KGDB is an order of magnitude faster than gStore for the optimal query (Q_dbp1). Thus, KGDB is superior to gStore in the selection operations. For the slowest query (Q_dbp4), the query time of the two systems is of the same order of magnitude.

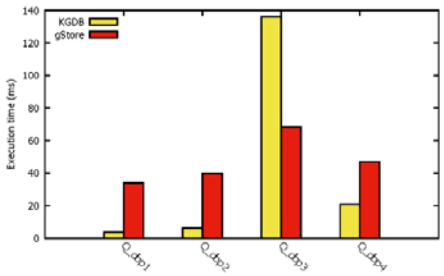


Figure 11. Experimental results of SPARQL query efficiency of KGDB and gStore on DBpedia

(2) Cypher query

Neo4j is a Cypher-based property graph database that obeys atomicity, consistency, isolation, and durability (ACID). It can process join operations faster.

As shown in Figure12, on LUBM datasets, KGDB spends less time than Neo4j for the three benchmark queries (Q1, Q3, and Q14); KGDB is nearly 70 times faster than Neo4j for the optimal query (Q3), and can reach the same order of magnitude of query time for the slowest query (Q2).

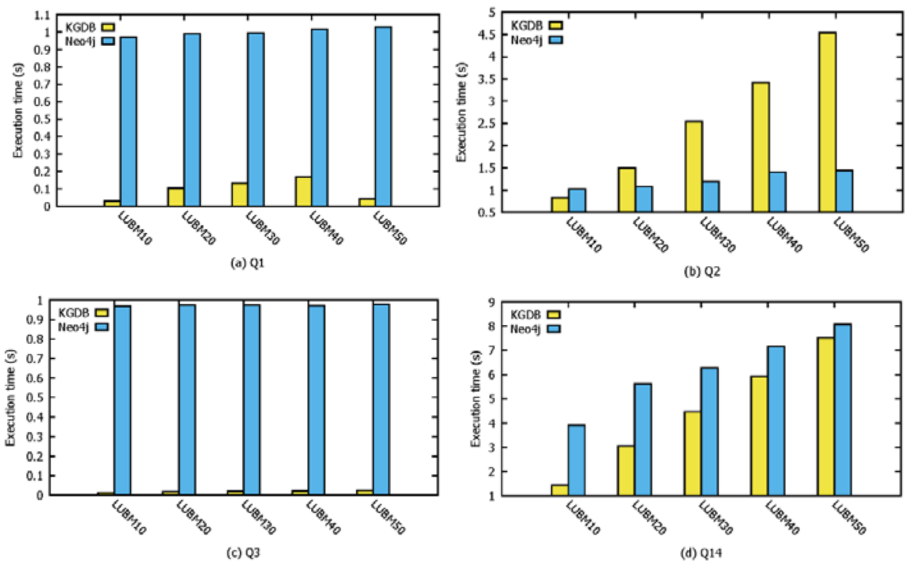


Figure 12. Experimental results of SPARQL query efficiency of KGDB and Neo4j on LUBMs

KGDB is slower than Neo4j for the most complicated triangular query (Q2). The query nodes are easily connected because of the storage scheme of Neo4j while KGDB, a relational database, has to conduct time-consuming JOIN operations. Nevertheless, KGDB surpasses Neo4j in more aspects: (1) KGDB originally supports the unified storage of RDF graphs and property graphs without a plug-in and calls for short time and less space than Neo4j, so it is more easier for KGDB to manage multiple knowledge graphs simultaneously. (2) KGDB supports both SPARQL and Cypher and realizes the interoperation between the two languages.

As shown in Figure13, on DBpedia, KGDB is faster than Neo4j for all the queries and can be two orders of magnitude quicker for the optimal query (Q_dbp3). Even for the slowest query (Q_dbp4), KGDB is 14 times faster than Neo4j.

With the data size increasing, the experimental results on LUBM datasets show us the following tendencies: Faster though it is, KGDB is less superior to Neo4j. Similar findings are proven for the most complex query (Q2). However, with the increase in data size, the query time of Neo4j rises at a lower rate than that of KGDB, but their difference is still of the same order of magnitude as before. However, due to semi-structured and sparse feature of real-world datasets, the increase in data size will give KGDB an advantage on DBpedia.

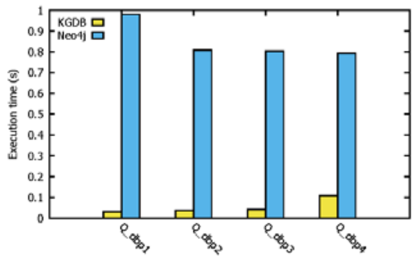


Figure 13. Experimental results of Cypher query efficiency of KGDB and Neo4j on DBp ia

6 Conclusions

In this paper, we develop KGDB: a knowledge graph database system with unified data model and query language.

(1) Unified storage scheme: KGDB can accomodate both RDF graphs and property graphs. Dictionary encoding is used in KGDB to save storage space. CS-based clustering is used to store untyped entities, so that semantically similar entities can be stored in the same relation table, increasing query efficiency.

(2) Interoperable grammar layer: SPARQL and Cypher are semantically aligned. In other words, the two languages can be used to obtain the same query results for the same knowledge graph, thus achieving interoperability.

(3) Unified semantic layer: SPARQL and Cypher can be transformed into query semantic trees in the form of relational algebra according to relevant rules.

Extensive experiments are carried out on real-world and synthetic datasets to verify the high efficiency of the unified storage scheme and query processing method. KGDB is generally more efficient than gStore and Neo4j in the storage management and query processing of knowledge graphs for real-world datasets. In addition, the efficiency of KGDB is of the same order of magnitude as that of gStore and Neo4j for synthetic datasets.

In this paper, we only discuss the knowledge graph management approaches of standalone systems. With the scale of data volume increasing, distributed knowledge graph management systems have been attracting increasing research efforts. In the future work, we will focus on the unified storage scheme and query processing method of knowledge graphs for the distributed environment.

References

[1] Wang X, Zou L, Wang CK, Peng P, Feng ZY. Research on knowledge graph data management: A survey. Ruan Jian Xue Bao/Journal of Software, 2019, 30(7): 2139–2174 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5841.htm [doi: 10. 13328/j.cnki.jos.005841]

[2] Zou L, Özsu MT, Chen L. gStore: A graph-based SPARQL query engine. The VLDB Journal, 2014, 23(4): 565–590.

[3] The Neo4j Team. The Neo4j manual v4.1. 2020. https://neo4j.com/docs/developer-manual/current/

[4] Dgraph Labs, Inc. The Dgraph homepage. 2020. https://dgraph.io/

[5] The HugeGraph Team. The HugeGraph manual. 2020. https://hugegraph.github.io/hugegraph-doc/

[6] Abadi DJ, Marcus A, Madden SR. Scalable semantic Web data management using vertical partitioning. In: Klas W, ed. Proc. of the 33rd Int'l Conf. on Very Large Data Bases. Vienna: VLDB Endowment, 2007. 411–422.

[7] Bornea MA, Dolby J, Kementsietsidis A. Building an efficient RDF store over a relational database. In: Ross K, ed. Proc. of the 2013 ACM SIGMOD Int'l Conf. on Management of Data. New York: ACM, 2013. 121–132.

[8] Moerkotte G, Neumann T. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. IEEE Trans. on Data Engineering, 2011, 984–994.

[9] Anagnostopoulos I, Mamoulis N, *et al*. Extended characteristic sets: Graph indexing for SPARQL query optimization. Proc. of the 2017 IEEE Int'l Conf. on Data Engineering (ICDE). California: IEEE, 2017. 497–508.

[10] Anyanwu K, Kim H, *et al*. Type-Based semantic optimization for scalable RDF graph pattern matching. Proc. of the 26th Int'l Conf. on World Wide Web. New York: ACM, 2017. 785−793.

[11] JanusGraph Authors. JanusGraph—Distributed graph database. 2020. http://janusgraph.org/

[12] TigerGraph. TigerGraph—The first native parallel graph. 2020. https://www.tigergraph.com/

[13] Zou L, Peng P. A survey of distributed RDF data management. Journal of Computer Research and Development, 2017, 54(6): 1213−1224 (in Chinese with English abstract).

[14] Wang TT, Rong CT, Lu W. Survey on technologies of distributed graph processing systems (in Chinese with English abstract). Ruan Jian Xue Bao/ Journal of Software, 2018, 29(3): 569−586. http://www.jos.org.cn/1000-9825/5450.htm [doi: 10.13328/j.cnki.jos. 005450]

[15] Harris S, Gibbins N. 3store: Efficient bulk RDF storage. In: Volz R, ed. Proc. of the 1st Int'l Workshop on Practical and Scalable Semantic Systems. Sanibel Island: CEUR-WS.org, 2004. 81−95.

[16] Pan Z, Heflin J. DLDB: Extending relational databases to support semantic Web queries. In: Volz R, ed. Proc. of the 1st Int'l Workshop on Practical and Scalable Semantic Systems. Sanibel Island: CEUR-WS.org, 2004. 109−113.

[17] Wilkinson K. Jena property table implementation. In: Smart PR, ed. Proc. of the 2nd Int'l Workshop on Scalable Semantic Web Knowledge Base Systems. Athens, 2006. 35−46.

[18] Abadi DJ, Marcus A, Madden SR. SW-Store: A vertically partitioned DBMS for semantic Web data management. VLDB Journal, 2009,18(2):385−406.

[19] Yuan P, Liu P, Wu B, *et al*. TripleBit: A fast and compact system for large scale RDF data. Proc. of the VLDB Endowment, 2013, 6(7):517−528.

[20] Neumann T, Weikum G. RDF-3X: A RISC-style engine for RDF. Proc. of the VLDB Endowment, 2008,1(1):647−659.

[21] Weiss C, Karras P, Bernstein A. Hexastore: Sextuple indexing for semantic Web data management. Proc. of the VLDBEndowment, 2008,1(1):1008−1019.

[22] Kim H, Ravindra P, *et al*. A semantics-aware storage framework for scalable processing of knowledge graphs on Hadoop. IEEE Trans. on Big Data, 2017:193−202.

[23] Sun W, Fokoue A, Srinivas K. SQLgraph: An efficient relational-based property graph store. In: Sellis T, ed. Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. New York: ACM, 2015. 1887−1901.

[24] The AgensGraph Team. Manual v1.0. 2020. https://bitnine.net/documentations/manual/agens_graph_developer_manual_en.html

[25] Chodorow K. MongoDB: The Definitive Guide: Powerful and Scalable Data Storage. O'Reilly Media, Inc., 2013.

[26] Blazegraph by Systap, LLC. Blazegraph. 2020. https://www.blazegraph.com/

[27] OpenLink Software. OpenLink virtuoso. 2020. https://virtuoso.openlinksw.com/

[28] Eclipse RDF4J. RDF4J. 2020. http://rdf4j.org/

[29] Neumann T, Weikum G. RDF-3X: A RISC-style engine for RDF. Proc. of the VLDB Endowment, 2008, 1(1): 647−659.

[30] Franz Inc. AllegroGraph. 2020. https://franz.com/agraph/allegrograph/

[31] Ontotext. GraphDB. 2020. http://graphdb.ontotext.com/

[32] Apache TinkerPop. TinkerPop3 documentation v.3.4.8. 2020. https://tinkerpop.apache.org/docs/3.4.8/reference/

[33] Callidus Software Inc. OrientDB-Multi-Model database. 2020. http://orientdb.org/

[34] S1CK. Cypher for apache spark. 2020. https://github.com/opencypher/cypher-for-apache-spark

[35] Gutierrez C, Hurtado CA, Mendelzon AO. Foundations of semantic Web databases. Journal of Computer and System Sciences, 2011, 77(3): 520–541.

[36] Francis N, Green A, Guagliardo P. Cypher: An evolving query language for property graphs. In: Das G, ed. Proc. of the 2018 Int'l Conf. on Management of Data. New York: ACM, 2018. 1433–1445.

[37] Guo Y, Pan Z, Heflin J. LUBM: A benchmark for OWL knowledge base systems. Web Semantics: Science, Services and Agentson the World Wide Web, 2005,3(2-3):158–182.

[38] University of Mannheim. DBpedia. 2020. http://wiki.dbpedia.org/About

Appendix

1. Query on LUBMs

(1) *SPARQL query*

① Q1

SELECT ?*X*
WHERE
{?*X* rdf:type ub:GraduateStudent.
?*X* ub:takesCourse
http://www.Department0.University0.edu/GraduateCourse0};

② Q2

SELECT ?*X*, ?*Y*, ?*Z*
WHERE
{?*X* rdf:type ub:GraduateStudent.
?*Y* rdf:type ub:University.
?*Z* rdf:type ub:Department.
?*X* ub:memberOf ?*Z*.
?*Z* ub:subOrganizationOf ?*Y*.
?*X* ub:undergraduateDegreeFrom ?*Y*};

③ Q3

SELECT ?*X*
WHERE
{?*X* rdf:type ub:Publication.
?*X* ub:publicationAuthor
http://www.Department0.University0.edu/AssistantProfessor0};

④ Q4

SELECT ?*X*, ?*Y*1, ?*Y*2, ?*Y*3
WHERE
{?*X*rdf:typeub:Professor.
?*X* ub:worksFor ⟨http://www.Department0.University0.edu⟩.
?*X* ub:name ?*Y*1.
?*X* ub:emailAddress ?*Y*2.
?*X*ub:telephone ?*Y*3};

⑤ Q5

SELECT ?*X*

WHERE

{?*X* rdf:type ub:Person.

?*X* ub:memberOf ⟨http://www.Department0.University0.edu⟩};

⑥ Q6

SELECT ?X **WHERE** {?*X* rdf:type ub:Student};

⑦ Q11

SELECT ?*X*

WHERE

{?*X* rdf:type ub:ResearchGroup.

?*X* ub:subOrganizationOf ⟨http://www.University0.edu⟩};

⑧ Q14

SELECT ?*X*

WHERE {?*X* rdf:type ub:UndergraduateStudent};

(2) *Cypher query*

① Q1

MATCH

(*x*:GraduateStudent)

□[takesCourse]→

(*y*:GraduateCourse

{*uri*:'http://www.Department0.University0.edu/GraduateCourse0'})

RETURN *x*;

② Q2

MATCH

(*x*:GraduateStudent) □[undergraduateDegreeFrom]→(*y*:University),

(*z*:Department) □[subOrganizationOf]→(*y*),

(*x*) □[memberOf]→(*z*)

RETURN *x*, *y*, *z*;

③ Q3

MATCH

(*x*:Publication)

□[publicationAuthor]→

(*y*:AssistantProfessor

{*uri*:'http://www.Department0.University0.edu/AssistantProfessor0'})

RETURN *x*;

④ Q14

MATCH

(*x*:undergraduatestudent)

RETURN *x*;

2. Query on DBpedia

(1) *SPARQL query*

① Q_dbp1

SELECT ?*a*

WHERE

{?*a* ⟨*uri*⟩ "http://dbpedia.org/resource/Alabama".

?*a* rdf:type ⟨AdministrativeRegion⟩};

② Q_dbp2

SELECT ?*a*

WHERE (?*a* rdf:type ⟨Disease⟩);

③ Q_dbp3

SELECT ?*a*
WHERE (?*a* rdf:type ⟨AdministrativeRegion⟩);

④ Q_dbp4
SELECT *
WHERE (?*a* rdf:type ⟨TimePeriod⟩);

(2) *Cypher query*

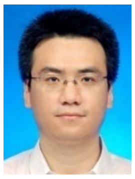
① Q_dbp1
MATCH
(*a*:AdministrativeRegion{*uri*:'http://dbpedia.org/resource/Alabama'})
RETURN *a*;

② Q_dbp2
MATCH (*a*:Disease) **RETURN** *a*;

③ Q_dbp3
MATCH (*a*:AdministrativeRegion) **RETURN** *a*;

④ Q_dbp4
MATCH (*a*:TimePeriod) **RETURN** *a*;

	LIU Bao-Zhu (1997–), master degree candidate, CCF student member, mainly engaged in data management of knowledge graphs.
	WANG Xin (1981–), PhD, professor, PhD supervisor, CCF outstanding member, mainly engaged in data management of knowledge graphs, graph database, and large-scale knowledge processing.
	LIU Peng-Kai (1998–), master degree candidate, CCF student member, mainly engaged in data management of knowledge graphs.
	LIU Si-Zhuo (1997–), master degree candidate, CCF student member, mainly engaged in data management of knowledge graphs.

	ZHANG Xiao-Wang (1980–), PhD, associate professor, PhD supervisor, CCF professional member, mainly engaged in knowledge graphs.
	YANG Ya-Jun (1983–), male, PhD, associate professor, mainly engaged in graph data management and graph mining.