A Maximum Semantic Reservation Mapping Method Based on Ontology-to-graph Database

Hongyan Wan[1,2], Huan Jin[1], Qin Zheng[1], Weibo Li[3], Junwei Fang[4]*

[1] School of Computer Science and Artificial Intelligence,Wuhan Textile University, China
[2] Engineering Research Center of Hubei Province for Clothing Information, Wuhan Textile University, China
[3] School of Economics, Wuhan Textile University, China
[4] School of Computer Science, Wuhan University, China
hywan@wtu.edu.cn, jh_0230@163.com, 5521084@qq.com, leewb@wtu.edu.cn, 961609701@qq.com

Abstract

Ontology is a core concept model in a knowledge graph which describes knowledge in the form of a graph. With the increase in knowledge graphs, the semantic relationships between concepts become more and more complex, which increases the difficulty of reserving its semantic integrity when storing it in a database. In this paper, we propose an ontology-to-graph database mapping method, which can reserve maximum semantic integrity and reduce redundant information simultaneously with high storage efficiency and query efficiency. In detail, the mapping method uses an anonymous class storage strategy to handle indefinite long nested structures, a multivariate functional relation storage strategy for multivariate semantic analysis, and an SWRL (Semantic Web Rule Language) storage strategy for disassembling inference structures. We develop an ontology-to-graph database prototype Neo4J4Onto to implement the mapping method. Experimental results show that our method achieves the maximum semantic integrity with the lowest complexity compared to the 6 baseline methods. Besides, compared to graphDB, Neo4J4Onto has better storage and query efficiency, and the concept models retrieved by Neo4J4Onto are more complete.

Keywords: Ontology storage, Graph database, Maximum semantic reservation, Semantic integrity, Query efficiency

1 Introduction

Ontology is the core of various knowledge graphs [1]. It uses a graph structure to represent knowledge, which makes the description of knowledge more concise, more standard, and easier to be shared [2]. Ontologyis being used in various fields [3]. The number and scale of ontology are increasing rapidly. The knowledge increase brings not only an increase inquantity but also an increase instructural complexity, which leads to many tough problems. These tough problems include how to efficiently and completely organize and store the semantic features in ontology, how to promote its query efficiency, how to increase its availability and applicability, etc.

Web-based ontology description languages are widely used in various knowledge graphs, such as RDF/RDFS [4] and OWL/OIL [5]. They can provide rich and convenient primitives to support semantic expression and reasoning [6]. However, the complete semantic information import of ontology needs to consider more complex mapping rules. Therefore, how to efficiently extract and completely store the semantic information in the RDF/OWL file in the database become a hot topic.

At present, there are three main types of database-based ontology storage solutions: relational database, triple database, and graph database [7-8]. Although relational databases [9-15] are easy to model and understand, with the increase of scale, the defects are gradually exposed. The inter-table connection involved in querying seriously reduces the system performance. The storage solutions based on a triple database [16-17] can maintain the ontology structure to a certain degree but it is relatively weak in the representation of complex semantic information.

The graph database-based storage solutions mainly rely on the advantages of graph structure, which matches the grid structure between concepts in the ontology. It can improve the storage and management efficiency, and the graph traversal and graph query algorithms in graph databases also improve the query efficiency, especially in large-scale data [18-19]. However, the existing methods have the problems of insufficient semantic preservation of ontology and have too much redundant information. More complex mapping rules need to be considered for the complete semantic information import of ontology.

Thus, we propose a semantic reservation-based ontology-to-graph database mapping method based on Neo4j (a typical graph database), which can not only reserve the maximum semantic integrity of ontology but also reduce the space complexity and promote storage efficiency. The main contributions of the mapping method are as follows:

(1) Storage solution for anonymous class: if the domain or the range in an axiom is declared by an anonymous class, namely, it is not declared by a named class with explicit URI, the nested structure with inter-classes operations and property restrictions in the anonymous class must be parsed correctly and efficiently.

(2) Storage solution for multivariate functions: as a

*Corresponding Author: Qin Zheng; E-mail: 5521084@qq.com
DOI: 10.53106/160792642023092405008

multivariate function describes the relationship among multiple classes (concepts), it is necessary to use multiple nodes and edges to represent the relations between them, and must guarantee the characteristics of the function at the same time.

(3) Storage solution for rules: as the head and body of a rule often contains several parameter assumptions and relation hypotheses, more efficient storage methods are needed to store classes, relations, functions, and individuals in large-scale rule applications.

The structure of this paper is as follows. Section 2 presents the related work about ontology storage methods based on a graph database. Section 3 presents the maximum semantic reservation method, including the mapping rules for ontology elements and the storage of rules. Section 4 presents the experiment and analysis, shows the comparison results of 7 ontology storage methods, and the storage space complexity analysis for the compared methods. The inference results and query efficiency comparison of graphDB and Neo4J4Onto are also shown in this section. Finally, the conclusions are represented.

2 Related Work

Many researchers have proposed various ontology storage methods based on graph databases [20-21]. The basic semantic information implied in RDF/OWL or SWRL can be directly transformed into the property graph model [22-23] in a certain way. Thus, the core contents (such as classes, instances, and relations) in the ontology can be directly mapped into graph databases. However, some complex semantic information in the ontology cannot be directly mapped to nodes and relationships. The semantics completeness of the large-scale upper-level ontology is essential to the lower-level knowledge's application in knowledge graphs [24].

The core and basic 4 kinds of elements in an ontology are named classes, individuals, object property relations, and data property relations. Mapping named classes and individuals to nodes and object property relations and data property relations to edges are commonly used practices [25-28]. Wang et al [29] mapped object property relations to nodes, and connect their domains and ranges by edges. Zhang et al [25] stored the axioms of classes and individuals through connecting the head and tail entities by edges. Bouhali et al [30] mapped a property axiom to the edge's property when the property relation is mapped to an edge to store the axioms describing the semantic information between property relations. Gong et al [31] mapped a property axiom and edge between two property relation nodes when the property relation is mapped to a node. For anonymous classes, some research stored the entire anonymous node in the form of sub-graphs, but the efficiency of the storage structure and the comprehensibility needs to be further improved [32-33]. For the rules, due to the complexity of its storage and application, most of the research did not support its storage [25-31].

There are some mature ontology storage systems, such as stardog [32] and graphDB [33], which are based on the RDF4J [34] framework. They can store the basic elements of ontology efficiently, but also generate much more nodes than the ontology has defined to store anonymous classes. Thus, the storage efficiency needs to be further improved. They only store binary function relations, which cannot meet the needs of multi-relational storage. Stardog can store the basic OWL inference rules, and graphDB can support the rule storage defined by SWRL, but their storage efficiency needs to be further improved.

All in all, current research cannot store the ontology semantic information completely and efficiently, especially for the storage of anonymous classes, multivariate functional relations and rules. In this paper, we propose a semantic reservation method for storing ontology based on Neo4J. It stores all the concepts, individuals (entities), relations, axioms, and rules in the ontology, and uses the structural characteristics of Neo4j to map the mesh structure knowledge of ontology, and stores procedures to implement rule-based reasoning. Our method concentrates on ensuring the maximum semantic integrity of the ontology and minimizing information redundancy.

3 Semantic Reservation Method

3.1 Overview of the Method

Most graph databases use the property graph model to create the graph, including Neo4j [35]. Compared with other models, the attribute graph model has richer expressive ability and can improve retrieval efficiency and save storage space. As shown in Figure 1, a property graph is a directed graph composed of vertex, edge, label, type, and property. A vertex is also called a node, and an edge is also called a relationship. All nodes have existed independently, and labels are set for nodes. Nodes with the same label belong to one group. Relationships are grouped by relationship types, and the same type of relationships belongs to the same set. A node can have zero or more labels, but the relationship must set only one relationship type. The relationship is directed. The two ends of a relationship are named starting node and the ending node. The directed arrows are used to identify the direction. The two-way relationship between the nodes is identified in two opposite directions. Nodes and relationships have their own set of properties. The set can be empty. If not, each property will be stored as a key-value pair. In the Neo4J property graph model, we can describe complex things and connections through the nodes which are connected by relationships. It also can describe various concepts and the complex relations between them.

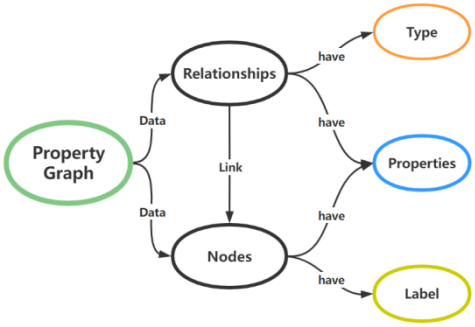


Figure 1. Property graph model

The property graph model has two core elements: nodes and relationships. Each node is an entity in the graph. It can contain enough properties in the form of key-value pairs, and can also be labeled. These labels are equivalent to a classification standard for specific knowledge domains. Each relationship can be directed, undirected, or named, and each relationship must have two corresponding nodes.

Ontology is composed of six basic modeling primitives <C, R, F, X, L, I>. For more details, please refer to [36].

The structure between the primitives in the ontology can be well matched with the graph structure of the property graph model as shown in Figure 2.

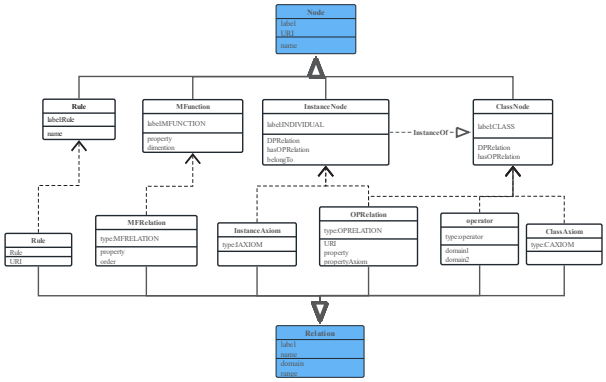


Figure 2. Mapping rules from ontology to graph database

The overall storage process is shown in Figure 3(a): firstly, the named classes and class axioms are parsed to construct the head and tail nodes in the graph database, then the relations are parsed to create the edges between those nodes according to Figure 3(b); next, the rules are stored in the storage process in a graph database to facilitate the application; finally, as the storage structures of individuals are determined by the instantiate classes, the individuals are parsed at the last step according to Figure 3(c). The mapping rules are as follows:

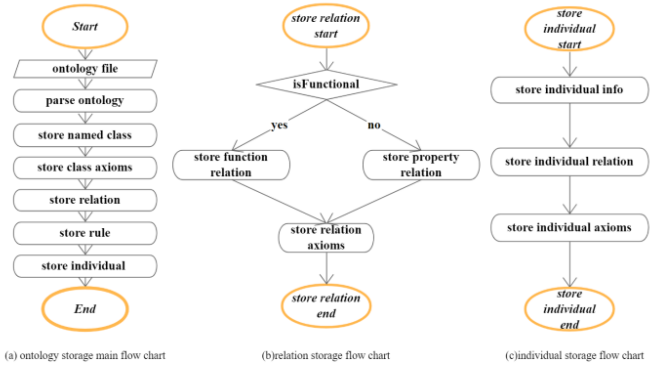


Figure 3. The overall ontology mapping method

3.1.1 Mapping Rules for Named Classes

A Named Class can be directly mapped to a node with the label CLASS, as shown in Table 1. According to the contents in the tag <owl: Class>, the class name and URI are retrieved and stored as properties of a node. The URI guarantees the uniqueness of the node. The data property relations that describe the named class are stored as the node's properties and processed by the relation storage algorithm.

Table 1. Tags of named class in OWL

Owl syntax	DL syntax	Example and comments
Owl: class	C	<owl:Class rdf: about=" #City"></owl:Class>

3.1.2 Mapping rules for Named Classes

An anonymous class does not have its own URI [37], but it always contains rich semantic information, which are always more complex than named classes. In property relations, Anonymous Classes are always used to define the domain concepts and/or the range concepts of the property relations. In class axioms, the parent class or the equivalence class of a named class is often represented by Anonymous Classes. Therefore, the creation of anonymous classes is extremely important to store and apply these kinds of property relations and class axioms.

As shown in Table 2, there are six ways to define anonymous classes inontology: in the form of intersection/ union/complement operation of classes (intersectionOf/ unionOf/ complementOf), in the form of enumerated individual (oneOf), in the form of value constraint and in the form of cardinality (Restriction). However, most Anonymous Classes are not just involving a single inter-class operation or a single property restriction. Their definitions may involve several anonymous classes, which result in an uncertainty length of the nested structure. Therefore, it is difficult to transform the semantic fragments (of Anonymous Class) with uncertain structure information into a sub-graph in the graph database.

To generate meaningful and shortest storage structures for Anonymous Classes, we proposed an anonymous class parsing algorithm for dealing with the indefinite nested structures. At first, the whole anonymous classesfragment is read into a string variable named anonimyFrag. Then, the semantic information is parsed from the innermost structure of the fragments to the outer layer one by one. When the labels defined in Table 2 are encountered, the corresponding anonymous nodes is created according to the semantic information described by anonymousclasses.

To improve the semantic expression capability of the created anonymous node, the naming rules are as follows:

(1) For an anonymous node generated by a class operation, its name is generated by combining the type of name of the operation (such as intersectionOf/ unionOf/ complementOf) with each class name involved in the operation.

(2) For an anonymous node generated by a property restriction, its name is generated by combining the property name with the range name of the property restriction.

Algorithm 1 shows the detailed steps for storing AnonymousClasses.

Table 2. Tags of anonymous class in OWL

Owl syntax	DL syntax	Example and comments
Class operator		
IntersectionOf (C1,…,Cn)	C1∩ … ∩Cn	<owl:intersectionOf rdf:parseType="Collection"> <rdf:Description rdf:about="#Budget Accommodation"/> <rdf:Description rdf:about="#Hotel"/> </owl:intersectionOf>
unionOf (C1, …,Cn)	C1 ∪ … ∪ Cn	<owl:unionOf rdf:parseType="Collection"> <rdf:Description rdf:about="#Adventure"/> <rdf:Description rdf:about="#"/> </owl:unionOf>
complementOf(C)	┐C	<owl:complementOf rdf:resource="#Sports"/>
oneOf(I1,…In)	{I1,…In}	<owl:oneOf rdf:parseType="Collection"> <rdf:Description rdf:about="#OneStarRating"/> <rdf:Description rdf:about="#ThreeStarRating"/> <rdf:Description rdf:about="#TwoStarRating"/> </owl:oneOf>
Property restriction		
Restriction(OP someValuesFrom C)	∃ OP,C	<owl:Restriction> <owl:onProperty rdf:resource="#hasActivity"/> <owl:someValuesFrom rdf:resource="#Hiking"/> </owl:Restriction>
Restriction(OP allValuesFrom (C))	∀ OP,C	<owl:Restriction> <owl:onProperty rdf:resource="#hasPart"/> <owl:allValuesFrom rdf:resource="#Beach"/> </owl:Restriction>
Restriction(OP hasValue (C))	∃ OP.{C}	<owl:Restriction> <owl:onProperty rdf:resource="#hasRating"/> <owl:hasValue rdf:resource="#ThreeStarRating"/> </owl:Restriction>
Restriction(OP minCaedinality (n))	≥n OP	<owl:Restriction> <owl:onProperty rdf:resource="#hasAccommodation"/> <owl:minCardinality rdf:data type="scheme a#nonNegativeInteger"/> 1</owl:minCardinality> </owl:Restriction>
Restriction(OP maxCardinality (n))	≤n OP	<owl:Restriction> <owl:onProperty rdf:resource="#hasAccommodation"/> <owl:maxCardinality rdf:data type="scheme a#nonNegativeInteger"/> 3</owl:maxCardinality> </owl:Restriction>

Algorithm 1. Storeanonymousclass

Input: String ontoURI:ontology URI
String anonimyFrag: the fragment of anonymous classes

Output: URI anonymousClassURI: the created anonymousClassURI for the whole anonymous fragment

Step1: Put all tags and URIs in anonimyFrag into a stack called fragStack; create an empty stack called tmpStack with NULL;
Step2: For each element tmp in fragStack:
Step2.1: If the tmp is URI, push it into tmpStack;
Step2.2: If the tmp is an operator, then
Step2.2.1: Create node for anonymousClass by the ontoURI and tmpStack, and return the unique identifier called anonymousClassURI;
Step2.2.2: For each element domainNodeURI in tmpStack, pop domainNodeURI and create an edge from the node identified by domainNodeURI to anonymousClass;
Step2.2.3: Push the anonymousClassURI into tmpStack;
Step2.3: If the tmp is a restriction, then
Step2.3.1: Create a node for anonymousClass by the ontoURI and tmpStack, and return the unique identifier called anonymousClassURI;
Step2.3.2: Pop the element restrictionURI and the next element rangeURI;
Step2.3.3: Create the restriction edge from anonymousClass to the node identified by rangeURI and assign the URI of restriction edge with restrictionURI;
Step2.3.4: Push the anonymousClassURI into tmpStack;
Step2.4: Pop the top element of fragStack
Step3: Pop the tmpStack and return the anonymousClassURI.

3.1.3 Mapping Rules for Relation

Table 3 shows the tags of relation in OWL. Relations in ontology contain relations between class and class, class and individual, and individual and individual. The relations involving individuals are handled by the individuals' mapping algorithm. In this section, only the relations between class and class are discussed. The relations between class and class can be divided into two types according to the range type:

One is data property relation. It is used to describe the unique feature of the class defined in the domain entity and its range is a value of a certain predefined datatype. To reduce the complexity of the graph database, a data property relation is mapped to the property of the class node in the graph database with <*relation*, *range*>, according to the information contained in <domain> in the fragment <DataProperty>.

The other is object property relation. It is used to describe the relationship between the class defined in <domain> and the class defined in <range>. Thus, an object property relation is mapped to an edge with the label OPRELATION from the class node defined in <domain> to the class node defined in <range>, according to the information contained in the fragment <objectProperty>. Besides adding the name and URI as the edge's attributes, property features including [transitive], [symmetric], [asymmetric], [functional], and [inverseFunctional] must be recorded as the edge's attributes too. It is worth noting that if the edge contains a [functional] feature, Algorithm 3 is needed.

Algorithm 2 shows the detailed steps for storing relations.

Table 3. Tags of relations in OWL

Owl syntax	DL syntax	Example and comments
Relation description		
owl:ObjectProperty	OP	<owl:ObjectProperty rdf:about="#hasActivity">
rdfs: domain (C_d)	OP (C_d, C_r)	<rdfs:domain rdf:resource="#Destination"/>
rdfs:range(C_r)		<rdfs:range rdf:resource="#Activity"/> </owl:ObjectProperty>
owl:DatatypeProperty	DP	<owl: DatatypeProperty rdf:about="#hasCity">
rdfs:domain(C_d)	DP (C_d, C_r)	<rdfs:domain rdf:resource="#Contact"/>
rdfs:domain(C_r)		</owl:DatatypeProperty>
rdf:type	OP type [character]	<owl:ObjectProperty rdf:about="#hasPart"> <rdf:type rdf:resource="owl#TransitiveProperty"/> </owl:ObjectProperty>

Algorithm 2. Storerelation

Input: Reasoneronto;// the result of ontology parsing

Output: Node nodes;// nodes updated according to all the DatatypeProperty relations defined in ontoURI

Edge edges; //edges created for all the ObjectProperty relations defined in ontoURI

Step1: For each datatypeProperty in onto:

Step1.1: Find the domain class node and add property according to the datatypeProperty;

Step1.2: If the datatypeProperty has axioms, call storeRelationAxiom to store relation axiom

Step2: For each objectProperty in ontoURI:

Step2.1: If the objectProperty is a functional property, call storeFunction to store functional property;

Step2.2: Otherwise, get domain class and range class according to the objectProperty;

Step2.2.1: If the domain or range is a fragment of anonymous class, call storeAnonymousClass and assign the returned URI to the domain or range;

Step2.2.2: Create an edge from the domain node to the range node, and add corresponding properties according to the features of the objectProperty;

Step2.2.3: If the objectPeorperty has axioms, call storeRelationAxiom to store relation axiom.

3.1.4 Mapping rules for Function Based on Multivariate Semantic Analysis

The function in ontology is a special kind of property relation. This kind of property relation has a functional features and involves two or more classes. When it only involves two classes, it is called Binary Functional Relation and the storage strategy is similar to the mapping rules for relations. The difference only lies in that the label BFUNCTION should be added to the nodes that have two classes. If it involves more than two classes, it is called Multivariate Functional Relation. It is obvious that maintaining the integrity of the n-ary functional relation (n>2) with the simple relation mapping rule is impossible.

As shown in Table 4, Multivariate Functional Relations (MFRs) are defined in OWL in two ways:

The first definition can transform a multivariate relation into multiple binary relations by reification [39-40]. To define an n-ary functional relation F (C_1, C_2,…, C_n), a total of n+1 classes must be defined firstly, namely the class identified by the name of MFR (C_F) and the classes C_1, C_2,…, C_n; and then, define n binary relations R_1, R_2,…, R_n to define the functional relationships between CF and C_1, C_2,…, C_n, namely, R_1: $C_F \rightarrow C_1$,…, R_n: $C_F \rightarrow C_n$; finally, use the tag <hasKey> to define the combination relation among the MFR and R_1, R_2,…, R_n.

Table 4. Tags of function in OWL

Owl syntax	DL syntax	Example and comments
Relation description		
owl:Functional-Property(F)	F	<owl:FunctionalProperty rdf:about="#hasMother"> <rdfs:domain rdf:resource="#Son"/> <rdfs:range rdf:resource="#Mother"/> </owl:FunctionalProperty >
HasKey (C(OP_1,…OP_n)| (DP_1,…DP_n))	OP(I1,I) and OP(In,I)→I1=I2; DP(I1,datatype) andDP(In,datatype)→I1=I2	<owl:Class ref:abouy="#ActivateAsIn"> <owl:HasKey> <owl:ObjectProperty rdf:resource="#hasName"/> <owl:ObjectProperty rdf:resource="#hasOccupation"/> <owl:ObjectProperty rdf:resource="#in"/> </owl:HasKey> </owl:Class>
rdf:List rdf:first rdf:rest		<owl:FunctionalProperty rdf:about="#calcuArea"> <rdfs:domain><rdf:List> <rdf:List><rdf:first rdf:resource="#Width"/> <rdf:rest rdf:resource="&rdf;nil"/></rdf:List> </rdf:rest></rdf:List></ rdfs:domain> <rdfs:range rdf:resource="#Area"/> </owl: FunctionalProperty >

As the idea of the reification follows the property graph model, the mapping algorithm is simple: create the class nodes according to the information in the <class> tag, generate the edges from the first-class node to the rest of class nodes, and name them according to the name of the binary functional relations. To ensure the integrity and accuracy of the semantics of MFR, adding the ordering property to each edge.

The second definition is to extend the functionality of the <domain> tag. The <domain> tag can contain multiple classes by combining with <List>, <First>, <rest> [41]. To define an n-ary functional relation F (C_1, C_2, …, C_n) in OWL, a total of n classes must be defined. The classes C_1, C_2, …, C_{n-1} is defined in tag <domain>, and C_n is defined in tag <range>. The relation's name is F. However, it is not following the property graph model. To ensure the integrity and accuracy of the semantics, it is necessary to reify the multivariate functional relation before storing it. We convert the multivariate functional relation into multiple binary relations. Specifically, the functional relation is stored as a node with the label MFUNCTION.A property to describe

the number of dimensions should be added to the node next. The classes involved in tag <domain>are connected to the function node, and the function node is connected to the class in the tag <range>. At the same time, the order property of the edge is recorded in the order of the function and the order property of the range node is 0.

However, in the property graph model, one edge only connects two nodes. To ensure the integrity and accuracy of the semantics of MFR, besides converting the MFR into multiple binary relations, the relations among the MFR and the binary relations must be kept. Specifically, in our algorithm, an MFR is stored as a class node (function node) with the label MFUNCTION and the dimension property. The classes involved in the <domain> tag are connected to the function node, and the function node is connected to the class in the <range> tag. At the same time, the order property of each edge is recorded according to the order of the function and the order property of the edge connected with the range node is 0.

Algorithm 3 shows the detailed steps for storing functions.

Algorithm 3. Storefunction

Input: OWL Relation relation; //the functional relation

Output: Created node and edge for function

Step1: Get all elements involved in the relation, the first element is called firstClass, and so on.

Step2: If the relationship has only two elements,

Step2.1: Create an edge from the firstClass to the secondClass;

Step2.2: If the firstClass is a class with <*HasKey*> tag, assign the count of elements involved in the tag to dimension. Add dimension property for the firstClass node with the value of the dimension.

Step3: Otherwise, the relation has more than two elements,

Step3.1: Create a node called MFnode and assign the relation's name to the node's name;

Step3.2: Assign the count of elements involved in the relation to dimension. Add dimension property to the node MFnode with the value of dimension;

Step3.3: Create an edge from the MFnode to firstClass, and add order property to the node firstClass with the value of the order. And do the same for other elements involved in the relationship.

An individual is mapped to a node with the label INDIVIDUAL. First, store the name, URI, and other properties in the individual node according to the <owl: NamedIndividual> tag. As the relation between a class and an individual indicates that the individual belongs to the class, which can be obtained by parsing the <type> tag, the mapping rule for the relations between an individual and its corresponding class is to create an edge named individualOf to connect the individual and the class to which it belongs. The relation between individual and individual corresponds to the object property relation between the class and the class, so we only need to connect the two individual nodes through an edge.

3.2 Mapping Rules for Relation

3.2.1 Mapping Rules for Class Axioms

Class Axiom is mapped to an edge in the graph database with the label CAXIOM, as shown in Table 5. As there are only 3 types of CAs (subClassOf/ equivalentClass/ disjointWith) to be considered, adding a name property to an axiom edge to identify its type is enough. A class axiom may come from the ontology itself or the alignment of two ontologies [42], adding a source property to each axiom edge to distinguish its origin is necessary for data provenance. If the domain or range class in the axiom is an anonymous class, the anonymous class creation algorithm must be called before the axiom edge is created.

Table 5. Tags of class axioms in OWL

Owl syntax	DL syntax	Example and comments
Class axioms		
Owl:subClassOf(C_1,C_2)	$C_1 \subseteq C_2$	<owl:Class rdf:about=" #Adventure"> <rdfs:subClassOf rdf:resource=" #Activity"/> </owl:Class>
Owl:equivalentClass(C_1,C_2)	$C_1 = C_2$	<owl:Class rdf:about="#AccommodationRating"> <owl:equivalentClass> <owl:oneOf rdf:parseType="Collection"> <rdf:Description rdf:about="#OneStarRating"/> <rdf:Description rdf:about="#TwoStarRating"/> <rdf:Description rdf:about="#ThreeStarRating"/> </owl:oneOf> </owl:equivalentClass> </owl:Class>
owl:disjointWith(C_1,C_2)	$C_1 \cap C_2 = \phi$	<owl:Class rdf:about="#RuralArea"> <owl:disjointWith rdf:resource="#UrbanArea"/> </owl:Class>

3.2.2 Mapping Rules for Individual

The mapping rule for Individual Axioms is similar to the mapping rule for Class Axioms, as shown in Table 6. An individual axiom is mapped to an edge between the individual nodes in the graph database with the label IAXIOM. At the same time, as the individual axiom may also be generated by the ontology alignment, the source property also needs to be added to the edge.

Table 6. Tags of individual axioms in OWL

Owl syntax	DL syntax	Example and comments
Individual axioms		
owl: sameAs (I_1,I_2)	$I_1 \subseteq I_2$	<rdf:Description rdf:about="#William_Jefferson_Clinton"> <owl:sameAs rdf:resource="#BillClinton"/> </rdf:Description>
wl: differentFrom (I_1,I_2)	$I_1 \neq I_2$	<rdf:Description rdf:about=="Cosi_fan_tutte"> <owl:differentFrom rdf:resource="#Don_Giovanni"/> </ rdf:Description >
owl: AllDifferent (I_1,…,I_n)	$I_1 \neq I_2 \ldots \neq In$	<owl:AllDifferent> <owl:distinctMembers rdf:parseType="Collection"> <rdf:Description rdf:about="#Don_Giovanni"/> <rdf:Description rdf:about="#Nozze_di_Figaro"/> <rdf:Description rdf:about="#Cosi_fan_tutte"/> </owl: distinctMembers> </owl: AllDifferent>

3.2.3 Mapping Rules for Property Relation Axioms

Table 7. Tags of property relation axioms in OWL

Owl syntax	Dl syntax	Example and comments
Property relation axioms		
owl:subPropertyOf (OP1,OP2) (DP1,DP2)	$OP_1 \subseteq OP_2$ or $DP_1 \subseteq DP_2$	<rdf:DatatypeProperty rdf:about="#aimChatID"> <rdfs:subPropertyOf rdf:resource="#nick"/> <rdfs:domain rdf:resource="#Agent"/> <rdfs:range rdf:resource="#Literal"/> </rdf:DatatypeProperty >
owl:equivalentProperty(OP_1,OP_2)| (DP_1,DP_2)	$OP_1 = OP_2$ or $DP_1 = DP_2$	<rdf: ObjectProperty rdf:about="#maker"> <owl:equivalentProperty rdf:resource="#creator"/> <rdfs:domain rdf:resource="#product"/> <rdfs:range rdf:resource="#Agent"/> </rdf: ObjectProperty >
owl:inverseOf(OP_1,OP_2)| (DP_1,DP_2)	$OP_1 \sim OP_2$ Or $DP_1 \sim DP_2$	<owl:ObjectProperty rdf:about="#hasActivity"> <owl:inverseOf rdf:resource="#isOfferedAt"/> <rdfs:domain rdf:resource="#Destination"/> <rdfs:range rdf:resource="#Activity"/> </owl:ObjectProperty>

Table 7 shows the tags of property relation axioms in OWL. The property relation axiom corresponds to the property relation, and the property relation is divided into data property relation and object property relation. When the axiom describes the relationship between two data property relations, it is necessary to find the domain and the range of the axiom which we called domain Property and range Property. Then, because the data property relation is stored as the node's property, the axiom is also stored as the node's property. So, we find the node according to the domain Property and add property <axiom, rangeProperty> to the node. In addition, we also add property <axiom, domainProperty> to the node which is the domain class of range Property.

When the axiom describes the relationship between two object property relations, it should be mapped to the property of the edges generated by the two object property relations in the graph database. And the two property names have opposite meanings (subPropetyOf && supPropertyOf).

3.3 Storage of Rules

Ontology rules are used to discover the implicit logical relationships in the ontology and check the compatibility of ontology and knowledge. SWRL is a language that semantically presents rules. The concept of rules in SWRL is evolved from RuleML [35] and combined with OWL ontology. SWRL can be seen as a combination of rules and ontology, in which the ontology can be used directly to describe the relationships between categories when writing rules. The rules in the ontology are divided into two parts:

The first kind is defining rules according to Description Logic, which is inferred by the semantic information between the tags specified in OWL. It is divided into class rules, individual rules, and property rules.

Common class rules include the following three types:

(1) **R1:** Subclass transitivity rules

(subClassOf(C1, C2) and subClassOf(C2,C3)) →subClassOf(C1,C3).

It means that if class C_1 is a subclass of C_2 and C_2 is a subclass of C_3, then C_1 is a subclass of C_3.

(2) **R2:** Subclass property inheritance rules

(subClassOf(C1,C2) and hasProperty(C2,A)) →hasProperty(C1,A).

It indicates that if class C_1 is a subclass of C_2 and C_2 has a property A, then C_1 also hasproperty A.

(3) **R3:** Subclass uncorrelation transfer rules

(disjointWith(C1,C2) and subClassOf(C,C1) →disjointWith(C,C2)).

It indicates that if classes C1 and C2 do not have intersection, and C is a subclass of C1, then C and C2 do not have an intersection either.

Common individual rules include the following two types:

(1) **R4:** Individual property inheritance rules

(individualOf(e,C) and hasProperty(C,A)) →hasProperty(e,A).

It indicates that if e is an individual of C and C has property A, then e also has property A.

(2) **R5:** Individual transfer property rules

(individualOf(e,C1) and subClassOf(C1,C2)) →IndividualOf(e,C2).

It indicates that if e is an individual of C1, C1 is a subclass of C2, then e is also an individual of C_2.

Common property rules include the following three types:

(1) **R6:** Property transitivity rules

(subPropertyOf(P1,P2) and subPropertyOf(P2,P3)) →subProperty Of(P1,P3).

It indicates that if P1 is a sub-property of P2 and P2 is a sub-property of P3, then P1 is also a sub-property of P3.

(2) **R7:** Property transfer attribution rules

(hasProperty (C, P1) and subPropertyOf (P1, P2)) →hasProperty (C, P2).

It indicates that if C has a property P_1 and P_1 is a sub-property of P_2, then C also has the property P_2.

(3) **R8:** Property inverse rule

(inverse (P1, P2) and inverse (P2, P3) → equivalentProperty (P1, P3)).

It indicates that if the properties P_1 and P_2 are mutually inverse, and P_2 and P_3 are mutually inverse, then the property P1 and P3 are equivalent properties.

The second kind is defining rules by SWRL. The SWRL language consists of four parts: Imp, Atom, Variable, and Built-in. As the rules defined by SWRL are various, defining a storage procedure for each rule is unpractical. To promote the degree of automation and generalizability, a graph based SWRL rule storage and application method are proposed as follows:

For each rule, map its parameters to nodes, and the relations between the parameters to edges. After that, distinguish the edges that belong to the rule head or rule body by adding a different type of label to the edge.

Algorithm 4 shows the detailed steps for storing rules.

Algorithm 4. Storerule

Input: Reasoner onto: the result of ontology parsing

Output: create nodes and edges for the rule

Step1: Parse the body of the rule:

Step1.1: For each variable in the body, create a node for the variable;

Step1.2: For each relation in the body, create an edge between variables involved in the relation with "ruleBody" type label;

Step2: Parse the head of the rule:

Step2.1: For each relation, create an edge between variables involved in the relation with "ruleHead" type label.

The rules defined by SWRL generally apply to the ontology of a specific domain while the structural information and quantity are uncertain. Through the rule mapping algorithm, all the SWRL rules can be converted into a property graph by decomposing the corresponding rule headers and bodies. A sub-graph reasoning obtains the structure of another subgraph. By combining the characteristics of the graph database, a rule inference machine can be implemented to execute the reasoning of all rule graphs in a unique way of "subgraph-condition-action".

4 Experiment and Analysis

In the experiment, the ontology is stored in Neo4J 3.4.0, and the programming language is Java. The experimental environment is listed below:

Operating system: ubuntu 16.04;

CPU: Intel Core i5-7300HQ CPU 2.50GHz;

Memory: 8.00 GB.

4.1 Integrity of Semantic Information

Ontology is disassembled into four categories: entities, relations, axioms, and rules. Entities are divided into Named ClassesI, Anonymous Classes (C*), and Individuals (I). Relations are divided into Object Property relations (OP), Data Property relations (DP), Binary Functional relations (BF), and Multivariate Functional relations (MF). Axioms are divided into Class aXioms (CX), Individual aXioms (IX), and Property Axioms (PX). The storage capabilities of 7 ontology storage methods are shown in Table 8.

TIe integrity of semanIic information should be guaranteed first for ontology storage. It can be seen from Table 9 that our method can store more complete ontology elements and has the best semantic retention than other methods. In addition to supporting the storage of basic elements of an ontology, it can also support the storage of anonymous classes, functional relationships, axioms, and rules.

Table 8. Storage capability comparison among 7 ontology storage methods

	Entity			Relation				Axiom			Rule
	C	C*	I	OP	DP	BF	MF	CX	IX	PX	L
Paper [25]	●	-	●	●	●	●	-	●	●	-	-
Paper [26-28]	●	-	●	●	●	-	-	-	-	-	-
Paper [29]	●	-	●	●	●	-	-	●	●	-	-
Paper [30-31]	●	-	●	●	●	-	-	●	●	●	-
Stardog [32]	●	●	●	●	●	●	●	●	●	●	-
graphDB [33]	●	●	●	●	●	●	-	●	●	●	●
Our method	●	●	●	●	●	●	●	●	●	●	●

Note: "-" indicates that the element cannot be stored, "●" indicates that the element can be stored.

4.2 Storage Space Complexity Analysis

The storage space complexity comparison is shown in Table 9.

To store named claIs (C) and individuals (I) in ontology, all methods need to create |C| and |I| nodes, including 2|C| and 2|I| properties (for storing the properties Name and URI of each node) in graph databases respectively. At the same time, the count of relations between the individuals and the classes is at least |I|.

Only the methods stated in [32-33] and our method can store anonymous classes (C*).

Suppose anonymous class fragments in a specific ontology involve m class operations and n restrictions, and the total number of concepts involved in the class operations is p. In our method, the number of generated anonymous nodes is $m + n$, and the number of node properties is $2(m + n)$. As an edge is generated each time a class is involved in a class operation or a property restriction, the number of edges generated for the anonymous class fragment is $p + n$, and the number of edge properties is $2(p + n)$. In graphDB/stardog, there will be p+2n+n nodes with p+2m+n node properties and p+m+n edges with p+m+n edge properties.

Our method can support the storage of both BFs and

Multivariate Functional relations (MFs). For BFs, they are stored as edges which are the same as OPs. Only |BF| edge and 2|BF| edge properties need to be added. For an MF involving n variants (MF_n), an additional node and ***n*** edges between this node and ***n*** variant nodes need to be generated. Suppose all the MFs include N variants. Then, N edges should be created.

For the storage of class axioms (CX) and individual axioms (IX), |CX| and |IX| edges need to be created. And as this kind of edge only needs name property, |CX| and |IX| edge properties need to be created respectively. For the storage of property axioms (PX), a property axiom is stored as an edge property. Since it is necessary to add a property to both the domain node and range node for the property axiom, 2|PX| edge properties need to be generated. For SWRL rules, only our method and graphDB support their storage. Assuming that the rule involves *l* variables and *t* relations, our methods will generate *l* nodes and *t* edges. Meanwhile, in graphDB, a total of at least *2l* nodes, 2l node properties, 2l+t edges and 2l+tedge properties will be generated.

In summary, the semantic reservation capability and the space complexity of our method are the best of all of the above methods.

Table 9. Storage space complexity analysis for the compared methods

C		Entity		Relation							
		C^*	I	OP	DP	BF&MF	CX	IX	PX	L	
[25]	N; NP	|C|;2|C|		|I|;2|I|		|DP|;|DP|					
	E; EP			|I|;|I|	|OP|;2|OP|	2|DP|;2|DP|	|BF|;2|BF|	|CX|;|CX|	|IX|;|IX|		
[26-28]	N; NP	|C|;2|C|		|I|;2|I|		|DP|;|DP|					
	E; EP			|I|;|I|	|OP|;2|OP|	2|DP|;2|DP|					
[29]	N; NP	|C|;2|C|		|I|;2|I|	|OP|;2|OP|	|DP|;|DP|					
	E; EP			|I|;|I|	2|OP|;2|OP|	2|DP|;2|DP|		|CX|;|CX|	|IX|;|IX|		
[30-31]	N; NP	|C|;2|C|		|I|;2|I|		0;|DP|					
	E; EP			|I|;|I|	|OP|;2|OP|			|CX|;|CX|	|IX|;|IX|	0;2|PX|	
Stardog [32]/ graphDB [33]	N; NP	|C|;2|C|	p+2m+n; p+2m+n	|I|;2|I|		|DP|;|DP|					2l;2l
	E; EP		p+m+n; p+m+n;	|I|;|I|	|OP|;2|OP|	2|DP|;2|DP|	|BF|;2|BF|	|CX|;|CX|	|IX|;|IX|	0;2|PX|	2l+t; 2l+t
Our Method	N; NP	|C|;2|C|	m+n; 2(m+n)	|I|;2|I|		0;|DP|	|MF_n|;2|MF_n|				l;l
	E; EP		p+n; 2(p+n)	|I|;|I|	|OP|;2|OP|		|BF|+N;2|BF|+N	|CX|;|CX|	|IX|;|IX|	0;2|PX|	t;t

Table 10. Comparison of single ontology storage

			Entity			Relation			Axiom			Rule	Total
			C	C^*	I	OP	DP	F	CX	IX	PX	L	
Travel.owl [38]	Elements		34	26	14	6	4	4	47	3	1	0	/
	graphDB	N;NP	34;68	58;58	14;28	-	-	0;0	-	-	-	0;0	106;158
	Our method	N;NP	34;68	26;52	14;28	-	0;4	0;4	-	-	-	0;0	74;153
	graphDB	E;EP	-	87;87	15;15	6;12	4;8	4;8	47;47	0;0	0;2	0;0	163;179
	Our method	E;EP	-	40;80	15;15	6;12	-	0;0	47;47	3;3	0;2	0;0	111;159
Step.owl [43]	Elements		21	7	0	16	3	6	12	0	1	2	/
	graphDB	N;NP	21;42	37;37	0;0	-	-	0;0	-	-	-	35;35	93;114
	Our method	N;NP	21;42	7;14	0;0	-	0;3	6;0	-	-	-	9;9	37;68
	graphDB	E;EP	-	65;65	0;0	16;32	3;6	6;12	12;12	0;0	0;2	39;39	141;168
	Our method	E;EP	-	8;16	0;0	16;32	-	0;0	12;12	0;0	0;2	8;8	44;70
Foaf.rf [44]	Elements		22	0	0	40	27	4	19	0	11	0	/
	graphDB	N;NP	15;37	0;0	0;0	-	-	0;0	-	-	-	0;0	15;37
	Our method	N;NP	22;44	0;0	0;0	-	0;27	4;0	-	-	-	0;0	26;71
	graphDB	E;EP	-	0;0	0;0	33;73	27;54	4;8	23;23	0;0	0;22	0;0	90;180
	Our method	E;EP	-	0;0	0;0	40;80	-	0;0	23;23	0;0	0;22	0;0	63;125

4.3 Ontology Import

As the papers in [25-31] did not provide codes, the following experiments are only carried out by comparing with the ontology storage tool graphDB and the prototype Neo4J4Onto is developed to store ontology according to our method.

Table 10 shows the number of elements contained in the three ontology files [38, 43-44] and the number of generated nodes (N), node properties (NP), edges (E), and edge properties (EP) by graphDB [33] and Neo4J4Onto.

There are many axiom descriptions in travel.owl involving anonymous classes, the number of nodes, and edges, But the storage space of our method is significantly smaller than that of graphDB. For step.owl, the number of anonymous classes is small, but it contains two SWRL rules. Our method still shows better performance than the graphDB. For foaf.rdf, there are multiple concepts derived from other ontology, which are not stored as nodes in graphDB. To keep the integrity and consistency of the semantic information, our method creates the corresponding nodes and edges for these entities, which is consistent with the protégé analysis results. Although the storage space is larger than graphDB, the semantic reservation capability of our method is better.

Besides single ontology import, Neo4J4Onto provides the function of batch ontology import. A unique node will be generated for the concepts with the same URI even if they are defined in different ontologies. In this way, the relationship between different ontologies will be established and the connections will lead to the transfer of properties, relations, and so on, which increases the description ability of the whole knowledge base.

4.4 Query in Database

The ontologies stored in the graph database (namely the ontology library) are always used as an upper conceptual model to construct, analyze and complete knowledge graphs. Finding various conceptual models to guide the information extraction of various unstructured data is a typical routine for the ontology library, the completeness of retrieved concept models and the query efficiency is vital. The inference capability determines the completeness of retrieved concept models. Thus, we analyze the inference capability among protégé, graphDB, and Neo4J4Onto as follows.

4.4.1 Inference Capability

Inference capability refers to the capability of retrieving conceptual models by applying axioms and rules. subClassOf, disjointWith, equivalentClass and hasproperty are the most frequently used axioms. As atoms were applied in various rules and used to complete the concepts contained in models and check their consistency, we selected them to evaluate the inference capability. The results are listed in Table 11.

Table 11. Inference capability analysis among three tools based on travel.owl [38]

Query	Owl	Protégé	graphDB	Neo4J4Onto
?sequivalentClass ?t	7	0	7	7
?s subClassOf ?t	30	32	78	45
?s disjointWith ?t	9	57	9	57
?s hasproperty ?t	6	-	69	69

(1) Inference capability of equivalentClass

In protégé, only the equivalent relation defined between named classes is inferred. But in graphDB and Neo4J4Onto, the equivalent relation defined between named classes, named class, and anonymous class, anonymous classes are all inferred. Thus, the results of graphDB and Neo4J4Onto are equal, and the result of protégé is 0 as the equivalentClassrelations defined in travel. Owl is between the named class and the anonymous class.

(2) Inference capability of subClassOf

In protégé, only the subclass relation defined between named classes is inferred. In Neo4J4Onto, the subclass relation defined between named classes, named class, and anonymous class, anonymous classes are all inferred. In graphDB, not only the subclass relation defined between different types of classes are inferred, the classes equivalent to each other are treated as the subclass of each other. Therefore, the result of protégé is the least, the result of graphDB is the best and ours is in the middle.

(3) Inference capability of disjointWith

In protégé, only the disjoint relation defined between named classes is inferred. In Neo4J4Onto, the disjoint relation defined between named classes, named class, and anonymous class, anonymous classes are all inferred. But in graphDB, no inference is carried out on the disjoint relation. Thus, the result of graphDB is the same as the one defined in the owl file and the results of protégé and Neo4J4Onto are the same as all the disjoint relations defined in travel.owl is between named classes.

(4) Inference capability of hasproperty

In graphDB and Neo4J4Onto, a class's property is retrieved according to its inheritance relationship. In protégé, it does not support this kind of inference. Therefore, the results on travel.owl are listed in Table 12. It is worth noting that the abundant properties attained from inference are vital to complete concept models which can be used to guide the information extraction further.

To sum up, our method performs well in reasoning and can deduce reasonable results. Figure 4 shows the *subclass* query result of the *Capital* concept in travel.owl by graphDB and Neo4J4Onto.

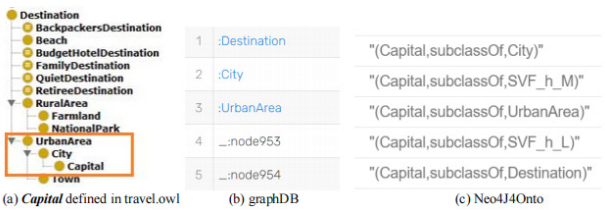


Figure 4. The inference results of graphDB and Neo4J4Onto

4.4.2 Query Efficiency

Different storage strategies result in different query efficiencies [45]. Table 12 lists the efficiency comparison of 4 kinds of query operations. Namely, (1) and (2) queries the subclass/superclass relationship between concepts; (3) and (4) queries about all subclass relationships in a single ontology and the whole database. The results indicate that Neo4J4Onto has a much better performance than graphDB.

The more complex the query, the more efficient Neo4J4Onto is.

Table 12. Query efficiency comparison between graphDB and Neo4J4Onto

Query	graphDB (ms)	Neo4J4Onto (ms)	Speed-up Ratio
(1)?subClassOfDestination	99	25	296%
(2) Capital subClassOf ?s	99	24	312.5%
(3) All subClassOf in travel.owl	235	26	803.8%
(4) All subClassOf in database	423	29	1358.6%

5 Conclusion

This paper proposes a mapping method of an ontology-to-graph database with maximum semantic reservation to meet the challenge of semantic incompleteness and storage information redundancy. The mapping method uses four strategies to ensure semantic integrity and high efficiency in storage and query. The mapping method proposed in this paper is implemented in Neo4J4Onto by comparing and analyzing the 6 baseline graph database storage methods. Experiment results show that our method has the best semantic integrity and better storage and query efficiency. The method has a general validity and can be bridged with popular ontology development tools such as protégé, which will be our further work in the future.

References

[1] L. Ehrlinger, W. Wöß, Towards a Definition of Knowledge Graphs, *Conference on Semantic Systems (SEMANTiCS)*, Leipzig, Germany, 2016, pp. 16-20.

[2] Y. Cui, L. Qiao, Y. Qie, Ontology Management and Ontology Reuse in Web Environment, *Challenges and Opportunity with Big Data: 19th Monterey Workshop*, Beijing, China, 2016, pp.122-130.

[3] Y.-C. Tian, D.-L. Jing, C.-C. Yang, Y.-X Chen, H.-J. Yang, An Ontological Approach for Architecture Design of a Smart Tourism System-of-Systems, *International Journal of Performability Engineering*, Vol. 16, No. 4, pp. 587-598, April, 2020.

[4] S. Decker, S. Melnik, F. V. Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann, I. Horrocks, The semantic web: The roles of XML and RDF, *IEEE Internet computing*, Vol. 4, No. 5, pp. 63-73, September-October, 2000.

[5] D.-L. McGuinness, F. V. Harmelen, OWL web ontology language overview, *W3C recommendation*, February, 2004, http://www.w3.org/TR/owl-guide/.

[6] X.-H. Wang, D.-Q. Zhang, T. Gu, H. K. Pung, Ontology based context modeling and reasoning using OWL, *Pervasive Computing and Communications Workshops*, Orlando, FL, USA, 2004, pp.18-22.

[7] Z. Ma, M.-A. Capretz, L. Yan, Storing massive resource description framework (RDF) data: a survey, *The Knowledge Engineering Review*, Vol. 31, No. 4, pp. 391-413, September, 2016.

[8] Z. Pan, T. Zhu, H. Liu, H. Ning, A survey of RDF management technologies and benchmark datasets, *Journal of Ambient Intelligence and Humanized Computing*, Vol. 9, No. 5, pp. 1693-1704, October, 2018.

[9] S. Wang, X. Zhang, A high-efficiency ontology storage and query method based on relational database, *International Conference on Electrical and Control Engineering (ICECE)*, Yichang, China, 2011, pp. 4253-4256.

[10] Z. Zhou, Y. Xing, A study on ontology storage based on relational database, *IEEE Conference Anthology*, China, 2013, pp. 1-5.

[11] R. Kwuimi, J.-V. Dombeu, Z. Tranos, An empirical analysis of semantic web mechanisms for storage and query of ontologies in relational databases, *International Conference on Advances in Computing and Communication Engineering (ICACCE)*, Durban, South Africa, 2016, pp. 132-136.

[12] J.-V. Fonou-Dombeu, R. Kwuimi, The Underpinnings of Ontology Storage in Relational Databases: An Empirical Study, *2018 International Conference on Advances in Big Data, Computing and Data Communication Systems (icABCD)*, Durban, South Africa, 2018, pp. 1-9.

[13] F. Zhang, Z.-M. Ma, W. Li, Storing OWL ontologies in object-oriented databases, *Knowledge-Based Systems*, Vol. 76, pp. 240-255, March, 2015.

[14] M.-S. Hema, R. Maheshprabhu, M.-N. Guptha, Data Access in Heterogeneous Data Sources Using Object Relational Database, *International Conference on Intelligent Information Technologies*, Chennai, India, 2017, pp. 23-33.

[15] S. P. Shantharajah, E. Maruthavani, A Survey on Challenges in Transforming No-SQL Data to SQL Data and Storing in Cloud Storage based on User Requirement, *International Journal of Performability Engineering*, Vol. 17, No. 8, pp. 703-710, August, 2021.

[16] T. Neumann, G. Weikum, The RDF-3X engine for scalable management of RDF data, *The VLDB Journal—The International Journal on Very Large Data Bases*, Vol. 19, No. 1, pp. 91-113, February, 2010.

[17] L. Zou, M. T. Özsu, *Graph-Based RDF* Data Management, *Data Science and Engineering*, Vol. 2, No. 1, pp. 56-70, March, 2017.

[18] C. Vicknair, M. Macias, Z. Zhao, X. F. Nan, Y. X. Chen, D. Wilkins, A comparison of a graph database and a relational database: a data provenance perspective, *Proceedings of the 48th annual Southeast regional conference*, Oxford, Mississippi, USA, 2010, Article No. 42.

[19] P. Pham, T. Nguyen, P. Do, Computing Domain Ontology Knowledge Representation and Reasoning on Graph Database, *Information Systems Design and Intelligent Applications*, Da Nang, Vietnam, 2017, pp. 765-775.

[20] I. Horrocks, P.-F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, SWRL: A semantic web rule language combining OWL and RuleML, *W3C Member submission*, September, 2004, https://www.w3.org/

Submission/SWRL/.
[21] C. Blankenberg, B. Gebel-Sauer, P. Schubert, Using a graph database for the ontology-based information integration of business objects from heterogenous Business Information Systems, *Procedia Computer Science*, Vol. 196, pp. 314-323, 2022.
[22] R. Angles, The Property Graph Database Model, *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management*, Cali, Colombia, 2018, pp. 23-30.
[23] N. Vanitha, C. R. R. Robin, D. D. H. Miriam, An ontology based cyclone tracks classification using swrl reasoning and svm, *Computer Systems Science and Engineering*, Vol. 44, No.3, pp. 2323-2336, 2023.
[24] P.-S. Sen, N. Mukherjee, Ontology-Based Data Modeling for NoSQL Databases: A Case Study in e-Healthcare Application, *SN Computer Science*, Vol. 4, Article No. 3, 2023, https://doi.org/10.1007/s42979-022-01405-5.
[25] H. Zhang, X. Hou, N. Li, A Storage Method of Ontology Based on Graph Database, *Fourth International Conference on Information Science and Cloud Computing (ISCC2015)*, Guangzhou, China, 2015, pp. 34-50.
[26] X.-W. He, Probing Optimisation of RDF Semantic Data Storage in Big Data, *Computer Applications and Software*, Vol. 32, No. 4, pp. 38-41 & 55, April, 2015.
[27] L.-H. Xiang, J-G. Gu, G. Wu, Distributed storage for RDF data based on graph database, *Computer applications and software*, Vol. 31, No. 11, pp. 35-39, November, 2014, DOI: 10.3969/j.issn.1000-386x.2014.11.009. (in Chinese)
[28] J.-H. Kang, Z.-X. Luo, Research on RDF data storage based on graph database Neo4j, *Information technology*, Vol. 6, pp. 115-117, June, 2015, DOI: 10.13274/j.cnki.hdzj.2015.06.030. (in Chinese)
[29] H. Wang, Q. Q. Zhang, W. W. Cai, Y. Jiang, Research on storage method for domain ontology based on Neo4j, *Application Research of Computers*, Vol. 34, No. 8, pp. 2404-2407, August, 2017, DOI: 10.3969/j.issn.1001-3695.2017.08.038. (in Chinese)
[30] R. Bouhali, A. Laurent, Exploiting RDF Open Data Using NoSQL Graph Databases, *Artificial Intelligence Applications and Innovations: 11th IFIP WG 12.5 International Conference (AIAI)*, Bayonne, France, 2015, pp. 8-10.
[31] F. Gong, Y.-H. Ma, W.-J. Gong, X.-R. Li, C.-T. Li, X. Yuan, Neo4j graph database realizes efficient storage performance of oilfield ontology, *PloS one*, Vol. 13, No. 11, pp. 12-14, November, 2018.
[32] Stardog, 2017, *Available: https://www.stardog.com/*.
[33] Ontotext, 2017, *Available: http://ontotext.com/products/graphdb/*.
[34] RDF4J, 2017, Available: *http://rdf4j.org/*.
[35] Neo4j, *http://neo4j.org/*.
[36] A. Gómez-Pérez, V. R. Benjamins, Overview of knowledge sharing and reuse components: *Proceedings of the IJCAI-99 workshop on Ontologies and Problem-Solving Methods (KRR5)*, Stockholm, Sweden, 1999, pp. 45-59.
[37] R. D. Virgilio, Smart RDF data storage in graph databases, *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Madrid, Spain, 2017, pp. 872-881.
[38] travel.owl, 2017, *http://protege.stanford.edu/ontologies/travel.owl/*, accessed in 2020.
[39] A. C. Kanmani, T. Chockalingam, N. Guruprasad, RDF data model and its multi reification approaches: A comprehensive comparitive analysis, *International Conference on Inventive Computation Technologies (ICICT)*, Coimbatore, India, 2016, pp. 1-5.
[40] A. Chebba, T. Bouabana-Tebibel, S. H. Rubin, Attributed and n-ary relations in OWL for knowledge modeling, *Computer Languages, Systems & Structures*, Vol. 54, pp. 183-198, December, 2018.
[41] H.-U. Krieger, C. Willms, Extending owl ontologies by cartesian types to represent n-ary relations in natural language, *Proceedings of the 1st Workshop on Language and Ontologies*, London, UK, 2015, pp. 1-7.
[42] Y. Kalfoglou, M. Schorlemmer, Ontology mapping: the state of the art, *The knowledge engineering review*, Vol. 18, No. 1, pp. 1-31, January, 2003.
[43] step.owl, 2017, *https://lov.linkeddata.es/dataset/lov/vocabs/step.*
[44] foaf.rdf, 2017, *http://xmlns.com/foaf/spec/.*
[45] A. Brek, Z. Boufaida, Enhancing Information Extraction Process in Job Recommendation using Semantic Technology, *International Journal of Performability Engineering*, Vol. 18, No. 5, pp. 369-379, May, 2022.

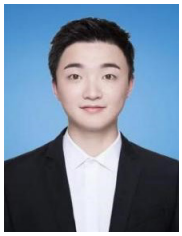
Biographies

Hongyan Wan received the Ph.D. degree from Wuhan University in 2021. She is currently a lecturer in the School of Computer Science and Artificial Intelligence at Wuhan Textile University. Her research interests include software engineering, natural language processing, machine learning, and intelligent algorithms.

Huan Jin was born in Daye City, Hubei Province. Her research interests include software requirements tracking and machine learning.

Qin Zheng is currently an associate professor in the School of Computer Science and Artificial Intelligence at Wuhan Textile University. Her research interests include software engineering and machine learning.

Weibo Li is currently an associate professor in School of Economics, Wuhan Textile University. His research interests include Internet of Things, cloud computing and big data, machine learning, network information retrieval, and intelligent algorithms.



Junwei Fang received the master degree from Wuhan University in 2020. His research interests include machine learning, requirement engineering, and intelligent algorithms.