Journal of Database Management
Volume 35 • Issue 1

RDF(S) Store in Object-Relational Databases

Zongmin Ma, Nanjing University of Aeronautics and Astronautics, China
https://orcid.org/0000-0001-7780-6473
Daiyi Li, Nanjing University of Aeronautics and Astronautics, China
Jiawen Lu, Nanjing University of Aeronautics and Astronautics, China
Ruizhe Ma, University of Massachusetts, Lowell, USA*
Li Yan, Nanjing University of Aeronautics and Astronautics, China

ABSTRACT

The Resource Description Framework (RDF) and RDF Schema (RDFS) recommended by World Wide Web Consortium (W3C) provide a flexible model for semantically representing data on the web. With the widespread acceptance of RDF(S) (RDF and RDFS for short), a large number of RDF(S) is available. Databases play an important role in managing RDF(S). However, there are few studies on using object-relational databases to store RDF(S). In this paper, the authors propose the formal definitions of RDF(S) model and object-relational databases model. Then they introduce the approach for storing RDF(S) in object-relational databases based on the formal definitions. They implement a prototype system to demonstrate the feasibility of the approach and test the performance and semantic retention ability of this prototype system with the benchmark dataset.

KEYWORDS

Object-Relational Databases, PostgreSQ, RDF(S), Storage

1. INTRODUCTION

The Semantic Web has been proposed by Tim Berners-Lee to provide a common framework for information sharing across multiple domains (Crasso *et al*., 2012). With the Semantic Web, data are provided with data semantic meaning (through metadata), and concepts and entities in the real world can be represented in a machine-readable and structured form. The Resource Description Framework (RDF) proposed by the World Wide Web Consortium (W3C) is a model of representing metadata of resources on the Web. RDF Schema (RDF(S)) as well as Web Ontology Languages (OWL) are the description of vocabulary semantics used in RDF datasets. RDF and RDF Schema (collectively known as RDF(S)) are the core of the Semantic Web. Nowadays, RDF(S) have been increasingly applied

DOI: 10.4018/JDM.334710 *Corresponding Author

in a wide range of Web-based application scenarios, such as semantic data integration (Arsic *et al*., 2019), semantic search (Xiong, Power and Callan, 2017; Zheng *et al*., 2019), semantic analysis of Big Data (Smiatacz, 2018; Shen, Hu and Tzeng, 2017), decision making (Rubio-Largo *et al*., 2017; Zhou *et al*., 2017) and so on. Currently, RDF(S) has become the de-facto standard of representing and handling data semantics. In particular, knowledge graphs (KGs) mostly adopt RDF mode to represent massive instances, and now are widely investigated and applied in diverse domains for the semantic and intelligent processing of massive data (Song *et al*., 2019).

With the rapid increase in the number of RDF(S) on the Web, it has become increasingly important to efficiently store massive amounts of RDF(S). The storage of RDF(S) (Ma, Capretz and Yan, 2016) often supports efficient queries of RDF data, mainly because the storage structure of RDF(S) not only directly determines the integrity of storage semantics, but also greatly affects its query efficiency (Ma *et al*., 2016; Ma, *et al*, 2018). At present, there have been many studies on RDF(S) storage methods, which can be roughly divided into the following three categories:

1) Memory-based storage (e.g., Sesame (Broekstra, *et al*., 2002) and BitMat (Atre, *et al*., 2008)). With this category of methods, memory space is directly allocated for RDF data and indexing technology is generally utilized for quick data process. Note that these methods are limited by the size of computer memory and are only suitable for storing a small number of RDF datasets;
2) Disk-based storage (e.g., YARS2 (Harth, *et al*., 2007) and System II (Wu, *et al*., 2009)). With this category of methods, the storage location is transferred from memory to hard disk. These methods meet the storage requirements of large-scale RDF datasets in space, but frequent reads and writes to disks greatly reduce storage performance;
3) Database-based storage (e.g., Jena-TDB (Wilkinson, *et al*., 2003), 4Store (Harris, *et al*., 2009), Virtuoso (Erling and Mikhailov, 2007), BigOWLIM/OWLIM-SE (Bishop *et al*., 2011), SPARQLcity/SPARQLverse[1], MarkLogic[2], and Clark and Parsia[3]). This category of methods uses database technology to store RDF data. In addition to some commercial systems, there are some developed prototypes such as RDF-3X (Neumann and Weikum, 2010), SW-Store (Abadi, *et al*., 2009) and RDFox[4].

With the mature technology and powerful data management capability of relational databases (RDBs), research on RDF(S) storage methods based on RDB have achieved some results (Ma, Capretz and Yan, 2016; Fan, Yan and Ma, 2020). However, due to the use of a two-dimensional table storage structure at the bottom of the RDB, which does not match the structure of RDF(S), the RDF(S) storage methods based on RDBs cannot effectively store the semantic information of RDF(S), resulting in incomplete storage data semantics and low query efficiency. To store massive RDF data, NoSQL (not only SQL) databases are applied to store RDF data (Cudre-Moroux *et al*., 2013). However, although NoSQL-based storage is highly efficient for massive RDF(S) data, it is recognized that there is no unified standard for NoSQL databases, and different databases use different query languages, each with its own advantages and disadvantages (Edwards, 2022). Considering the cost, familiarity and technical maturity, for the storage of non-massive RDF data, traditional databases rather than NoSQL databases are still the first choice due to the mature theoretical basis and powerful data management capabilities.

Object-relational databases (ORDBs) are based on RDBs and combine object-oriented features. Therefore, they not only support the integrity constraints and SQL standards of RDBs, but also utilize object-oriented features to handle complex data relationships. At present, although ORDBs are widely applied in various domains such as geographic information management (Ackere *et al*. 2019), software engineering (Gregory, 2019), multimedia (Khanduja and Chkraverty, 2019) and other fields, there is relatively little research on using ORDBs to store RDF(S). In (Alexali *et al*., 2001), a tool called RDF Suite was implemented based on an ORDB, which can be used for RDF(S) storage and querying. This RDF(S) storage method achieves the separation of RDF(S) schema information

and data information, mainly using four data tables: class, subClass, property, and subProperty to store RDF(S) schema information, and then connecting instance tables based on inheritance. This method is more suitable for scenarios with more pattern types and less instance data. In (Astrova *et al*., 2008), a method for storing ontology in an ORDB was proposed. This method directly stores ontology as an object in the database, but does not store semantic relationships such as attributes, resulting in incomplete semantic information stored in the database. The Sesame tool (Broekstra *et al*., 2002) is also an RDF(S) storage tool based on an ORDB. This tool creates a data table for each class (the class table only contains a field to record the URI), which is used to record all instance information of that class. Meanwhile, the tool also creates a data table for each attribute (the attribute table contains two fields: subject and object), which is used to record the instance information of the attribute. Class tables and property tables are created according to corresponding inheritance relationships. The experimental results show that although this method can effectively maintain the semantic information of RDF(S), its query efficiency is not as good as that of relational databases.

In summary, RDBs cannot fully preserve the semantic information of RDF(S) due to a natural mismatch between their structural patterns and RDF(S). Meanwhile, unreasonable storage structure design can lead to situations such as excessive internal or external connections, which can lead to a decrease in query efficiency. Due to the lack of unified standards and query languages in object-oriented databases (OODBs), the technological development is still not mature enough. Although it can better express RDF(S) semantic information, it is difficult to achieve good query results, and its usability in actual production is low. The structure of ORDBs has a certain similarity to that of RDF(S), which can handle complex data relationships, has stable performance, and high commercial value. Therefore, it is currently a good choice for storing RDF(S). However, existing solutions for storing RDF(S) based on ORDBs still have the following issues and challenges:

1) Unable to store RDF(S) semantic information well. Some storage schemes only store RDF triplet data, with little consideration given to the structural semantic information contained in the RDF(S), resulting in incomplete RDF(S) storage information.
2) Unreasonable storage structure design. A large number of null values appear in the data table, and there may be too many internal or external connections when querying the database, which seriously affects query efficiency.
3) Some studies only utilize the theoretical methods of storing RDF(S) based on OEDBs without implementing corresponding system tools, and have not tested the performance of the proposed methods.

In this paper, we advocate to apply the ORDBs to persist RDF triples as well as RDF Schema data. The contributions of this work mainly include three aspects. 1) We formally define the object-relational database model and the RDF(S) model. The structure and semantics of RDF(S) and RDBs are comprehensively summarized, which lays a theoretical foundation for the storage of complete semantics in RDF(S); 2) Based on the formal definitions of these two models, we propose some rules for mapping RDF(S) into the ORDBs. These mapping rules can effectively reduce null values in data tables, avoid a large number of data table connections during queries, and support inference queries on RDF(S) stored in the ORDBs; 3) We develop a prototype system named RDF2ORDB to verify the feasibility of our storage approach. By comparing with existing RDF(S) storage methods, we demonstrate that the proposed method can effectively retain the semantic information of RDF(S) and achieve better query efficiency.

The rest of this paper is organized as follows. Section 2 gives an overview of the research status of RDF(S) storage. The formal definitions of RDF(S) and ORDBs are introduced in Section 3. In Section 4, we propose a storage framework for storing RDF(S) in the ORDBs. We implement and verify our approach in Section 5 and Section 6 concludes this paper.

2. RELATED WORK

RDF(S) storage has been studied in a variety of contexts. We can divide RDF(S) storage into two categories, which are file-based storage and database-based storage, as shown in Figure 1.

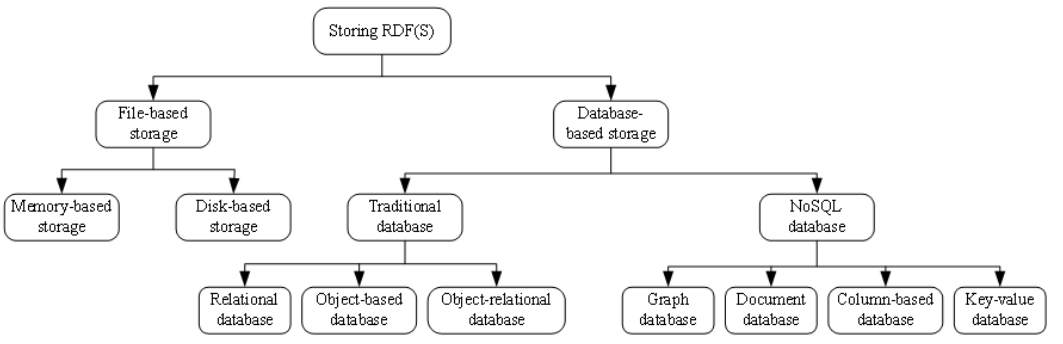
File-based storage is built directly on the file system. This method is easy to use and extend. However, when the scale of RDF(S) keeps increasing, file-based storage is not convenient for data maintenance and the query efficiency is not ideal. Therefore, this method is only suitable for storing small-scale RDF(S) and operations such as updates and queries do not occur frequently.

Relational databases are widely used and have good commercial value, which is the preferred method for storing RDF(S). Note that the relational databases store data in the form of two-dimensional table, which do not match the structure of RDF(S). Many methods have been proposed to decompose RDF(S) into the form that can be stored by relational databases (RDBs). At present, methods for storing RDF(S) based on RDBs can be divided into three categories:

- *vertical storage* (Broekstra, *et al.*, 2002; Wilkinson, *et al.*, 2003; Neumann and Weikum, 2010; Harris and Gibbins, 2003), which is also known as triple stores such as Sesame[5];
- *horizontal storage* (Agrawal, *et al.*, 2001; Bornea *et al*., 2013) like SW-Store[6];
- *property/type storage* (Pan and Heflin, 2003; Levandoski and Mokbel, 2009) like Jena[7].

1) **Vertical storage**, also known as triple storage, mainly stores RDF triples and cannot store RDF Schema schema information. An RDF instance can be expressed in the form of a triple, which consists of three parts: *subject*, *predicate*, and *object*. The vertical storage method directly stores RDF triplets in sequence in a data table containing the *subject*, *predicate* and *object* fields. This method has good scalability, can directly add data to the data table, and the parsing of RDF is also very simple. However, it cannot store RDF Schema information and cannot use RDF Schema for inference, resulting in poor semantic expression ability. In addition, querying data tables involves a large number of self-join operations, and as the complexity of SQL language increases, query efficiency will significantly decrease.
2) **Horizontal storage:** The basic idea of horizontal storage is to store all properties of RDF(S) as fields in a table, and a row of data in the data table is a complete instance. Compared to vertical storage, horizontal storage can effectively avoid a large number of self-join operations when querying RDF(S), and the query statement is relatively simple. In (Agrawal *et al*., 2001), it was achieved to present vertically stored data to users in the form of a horizontal table, allowing users to write query statements based on the horizontal storage mode without the need for complex SQL statements. In (Bomea *et al*., 2013), a table structure was designed that includes an *entry* field and multiple *predi* and *value*

Figure 1. RDF(S) storage approaches



fields. Each instance in RDF stores a *subject* in the *entry* field, and all predicates and objects owned by that instance are stored in multiple *predi* and *value* fields, respectively. This method provides an algorithm to map the same *predicate* from different instances to the same *predi* field in the data table for storage. Although the horizontal storage model is simple, instances of different classes in RDF(S) have different properties, and using horizontal storage can result in a large number of null values. In addition, the properties of an instance may have multiple property values, and the horizontal storage mode cannot solve the problem of multi valued properties.

3) **Attribute storage** can be seen as *predicate* oriented storage. This method creates a data table for each property in RDF(S) and divides RDF triplets into different property tables based on the properties. In (Pan and Heflin, 2003), it was achieved to create a property table while also creating a data table for storing instance information for all classes, and using views to store RDF schema information. However, this method considers incomplete semantic information and the efficiency of obtaining class hierarchy information through views is not ideal. In (Levandoski and Mokbel, 2009), triples are divided based on *predicates* and stored in different property tables. although joins between different tables are faster than self-join operations, the efficiency of query will decrease with the increase of query properties.

Due to the natural mismatch in structure between RDBs and RDF(S), the RDBs are not ideal for storing RDF(S). Object-oriented databases (OODBs) integrate the powerful modeling capabilities of the object-oriented paradigm into database model and store data in the form of objects, which are similar to RDF(S) in structure (Bagui, 2003). In (Chao, 2007), an object-oriented data model is presented for storing data extracted from RDF(S) and a generic API to support basic RDF(S) query operations. In (Zhang, *et al*, 2015), a method of using RDBs to store ontology is proposed, which could better retain semantic information expressed by ontology. Unlike the popular RDBs that are very mature, the OODBs do not have unified standards of the implementation and query. In the terms of scalability, fault tolerance, transaction support and other fields, the OODBs are far behind the RDBs in a practical view.

Recently, NoSQL databases have emerged as a commonly used infrastructure for managing Big Data. Particularly, NoSQL databases have been applied to store massive RDF(S) (Cudre-Mauroux *et al*., 2013). Depending on concrete data models for RDF data storage, the NoSQL-based stores of RDF data are categorized into *key-value stores*, *column-family stores* (e.g., HBase in (Khadilkar, *et al*, 2012; Franke *et al*., 2011)), *document stores* (e.g., CouchDB in (Stefani and Hoxha, 2018) and MongoDB in (Michel, *et al*, 2019)) and graph stores (e.g., Neo4j in DBpedia4neo[8]). Unlike traditional databases, NoSQL databases include several types databases, which lack unified standards and query languages. Their syntax for data manipulation varies depending on the types of NoSQL databases. In addition, NoSQL-based data management still needs improved. For example, the distributed nature of NoSQL databases enables faster data availability, but it may make ensuring data consistency even more difficult; queries may not always return updated data and may return inaccurate information (Edwards, 2022).

With ORDBs, a storage tool named RDF Suit is implemented in (Alexaki *et al*. 2001), which creates four table named *class*, *subclass*, *property* and *subproperty* for storing RDF Schema. Note that this approach pays attention only to storing more schema information rather than massive instance (triple) data. The method proposed in (Astrova and Kalja, 2008) is actually designed to store ontologies in ORDBs rather than RDF triples. Sesame (Broekstra, *et al*, 2002) supports storing RDF(S) with ORDBs. It creates tables for each class and property and stores instances of the corresponding class and property. As we show in the experiment of this paper, the query efficiency of this tool is not good enough.

3. FORMAL DEFINITIONS OF ORDBS AND RDF(S)

In this section, we propose formal definitions of RDF(S) model and ORDBs model. The definitions provide an overview of major features of RDF(S) and ORDBs, which are very helpful to illustrate the storage method proposed in Section 4.

3.1 Formal Definition of RDF(S) Model

The basic idea of RDF(S) is that all information stored in it is regarded as resources. Each resource in RDF(S) is identified with a unique International Resource Identifier (IRI). As the basic composition unit of RDF(S) model, RDF statement is comprised of subject, predicate and object, which can be abbreviated as (s, p, o). In a statement "Mathematic is learned by Student Mike", for example, "Mathematic", "taught by" and "Student Mike" are subject, predicate and object, respectively. A set of RDF triples can be presented as an RDF(S) graph. The nodes of an RDF(S) graph are subject or object and its directed edges are predicate.

For the reason that RDF only provides simple descriptions about resources and their values, specific properties or special relationships between different resources can be expressed by RDF Schema. The important components of RDF Schema include *rdfs:Class, rdf:Property, rdfs:Datatype, rdfs:subClassOf, rdfs:subPropertyOf, rdf:type, rdfs:domain, rdfs:range* and so on. The features of RDF Schema are similar to object-oriented model.

To retain RDF(S) semantic information more completely, we propose a formal definition of RDF(S) model as follows:

Definition 1. RDF(S) model can be defined as a three-tuple {*ConceptSet*, *AxiomSet*, *InstanceSet*}.

(1) *ConceptSet* = *ClassSet*∪*PropotySet*∪*DatatypeSet* is a finite set of basic concepts, in which *InstanceSet* represents a finite set of all classes, *PropotySet* represents a finite set of all properties, and *DatatypeSet* represents a finite set of all data types. Also, *PropotySet* = *Data-PropotySet*∪*Object-PropotySet*, where *Data-PropotySet* represents datatype properties and *Object-PropotySet* represents object properties.
(2) *AxiomSet* = *ClassAxiom*∪*PropertyAxiom* is a finite set of axioms of RDF(S), which *ClassAxiom* is a finite set of axiomatic declaration on classes and *PropoertyAxiom* is a finite set of axiomatic declaration on properties.
(3) *InstanceSet* is a finite set of instance resources and *InstanceSet* = *I*∪*B*∪*L*. Here *I* represents the finite set of all resources, *B* represents the finite set of all empty nodes and *L* represents the finite set of all literal resources. RDF also can be expressed as a triple $t = <s, p, o>$, where $s \in I \cup B$, $p \in I$, $o \in I \cup B \cup L$.

Based on the formal definition above, the main syntax of RDF(S) used in this paper is shown in Table 1.

To understand the structural information of RDF(S), we provide an example of RDF(S) data.

As shown in Figure 2. Nodes *Person*, *School*, *Student* and *University* are two classes. Nodes *doctorDegreeFrom* and *degreeFrom* are object properties. The lass *Student* is a subclass of the *Person* class and this semantic relationship belongs to *CAxiom*. The property inheritance relationship between *degreeFrom* and *doctorDegreeFrom* belongs to *PAxiom*. Node *Mike* is the subject of the triple and node *Stanford University* is the object of the triple. *doctorDegreeFrom* is the predicate of the triple. The property *doctorDegreeFrom* has the domain "Student" and the range "*University*", meaning a mapping described by the property from the domain to the range.

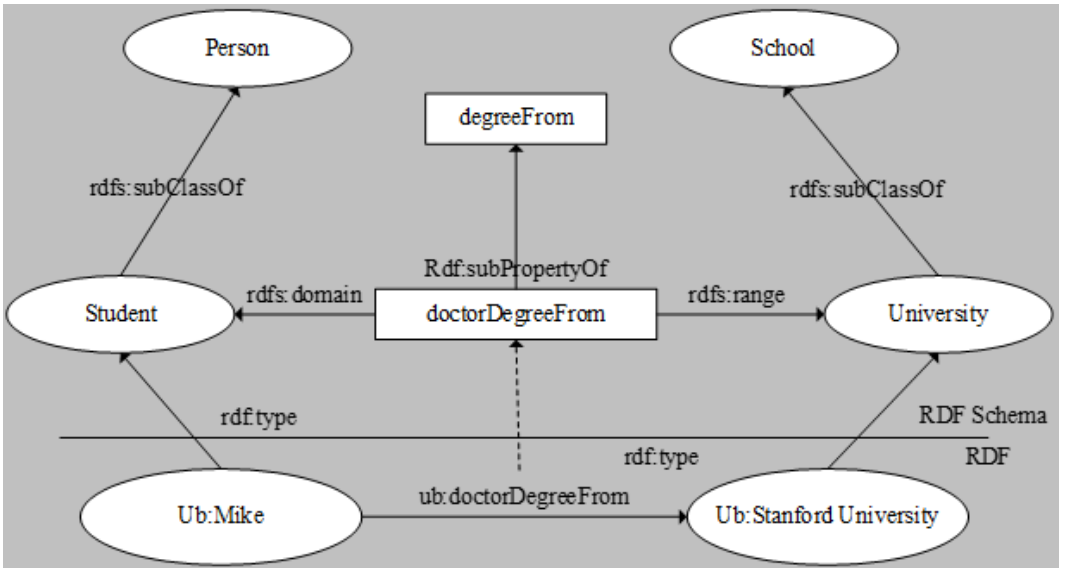
3.2 Formal Definition of Object-Relational Database Model

The ORDBs combine the advantages of object-oriented databases and relational databases to model and encapsulate the stored data in a more complex way. It not only supports object-oriented features

Table 1. Main syntax of RDF(S)

Syntax	Description
type(x) ∈ClassSet∪PropertySet	x is the resource referenced by IRI. type(x) is the class or property to which the resource belongs.
type(L)∈ DatatypeSet	L stands for literal resources. type(L) represents the datatype of the literal resource.
dom(p) ∈ ClassSet	p represents a property and dom(p) indicates the domain of the property is a class.
ran(p) ∈ DatatypeSet∪ClassSet	p represents a property and ran(p) indicates that the range of the property can be either a class or a primitive datatype.
pro(c) ⊆PropertySet	pro(c) represents the collection of all properties of the class C.
parent(c)⊆ClassSet	parent(c) represents the collection of all parents of the class C.
sco(c_1, c_2) ∈ClassAxiom	c_1 is a subclass of c_2
spo(p_1, p_2) ∈PropertyAxiom	p_1 is a subproperty of p_2
sub(t)∈s	t is an instance, sub(t) is the subject of the instance.
pre(t)∈p	t is an instance, pre(t) is the predicate of the instance.
obj(t)∈o	t is an instance, obj(t) is the object of the instance.
name(IRI)	name(IRI) represents the namespace of IRI.

Figure 2. An example of RDF(S)



such as inheritance, composite datatype customized by users, but also supports updating and querying databases with SQL statements (Auzi *et al*., 2018).

Since the ORDBs still adopt the two-dimensional table structure to store data, which is as same as relational databases, ORDBs should also meet the integrity constraints. The integrity constraints mainly include *not null constraints*, *unique constraints*, *primary key constraints*, *foreign key constraints*. The *not null constraint* is used to specify that a field does not have a null value. The *unique constraint* is used to specify that data in a field or set of fields in a data table is unique across all rows. The *primary key constraint* is generally used as a unique identifier of the row data.

The field to which the *primary key constraint* is added must satisfy both the not null and unique constraints. The *foreign key constraint* maintains referential integrity between two associated tables. The *foreign key constraint* specifies that the value of the field must be the same as some data value in the associated data table.

At present, there is still no standard formal definition of ORDBs. In order to generalize characteristics of the object-relational database, this paper puts forward the formal definition of object-relational database as follows:

Definition 2: An object-relational database can be defined as a four-tuple (*Basic*, *Cons*, *Inh*, *Ins*).

Basic represents a finite set of basic concepts in an ORDB. *Basic* = *Tab*∪*Col*∪*Dtype*. Tab is a finite set of all tables in an object-relational database. *Col* is a finite set of fields in a data table. *Dtype* is a finite set of datatypes. *Dtype* = *Ptype*∪*Ctype*. *Ptype* is a collection of primitive datatypes. *Ctype* is a collection of user-defined composite datatypes.

Cons is a collection of constraint relationships in an object-relational database. *Cons* = *Pcons*∪*Fcons*∪*Ucons*∪*Ncons*. *Pcons* is a set of *primary key constraint*s. *Fcons* is a finite set of *foreign key constraint*s. *Ucons* represents a finite set of *unique constraints*, and *Ncons* is a finite set of all *not null constraint*s.

Inh is finite set of inheritance relationships in an object-relational database. *Inh* = *SInh*∪*MInh*. *SInh* is a collection of single inheritance relationships and *MInh* is a collection of multi-inheritance relationships.

Ins is a collection of all instances in a database.

Based on the formal definition of ORDBs, we define its abstract syntax. Table 2 shows the main abstract syntax for the ORDBs. Note that, in this paper, we apply the ORDBs to persist RDF(S) and do not mention some advanced features of the ORDBs (e.g., multiple inheritance and method of ORDB model). Ones can refer to (Auzi, A. *et al*., 2018) for more details.

4. STORAGE OF RDF(S) IN ORDBS

In this section, we propose an approach to store RDF(S) in the ORDBs based on the formal definitions of two models proposed in Section 3. Specifically introduced the general architecture of the storage pattern and storage rules.

Table 2. Main syntax of ORDBs

Syntax	Description
c∈Col, table(c) ∈ Tab	c is a field and table(c) means the table to which field c belongs.
t∈Tab, col(t) ∈Col	t is a table and col(t) represents all fields contained in table t.
c∈Col, type(c) ∈Dtype	c is a field and type(c) means the data type of c.
i∈Ins, table(i)∈Tab	i is an instance and table(i) represents the table to which i belongs.
i∈Ins, column(i)∈Col	i represents the data instance and column(i) represents the field in which the data resides.
PK(t, c)∈Pcons	Primary key of table t contains field c.
FK(t, c)∈Fcons	Foreign key of table t contains field c.
FK(t_1, c_1, PK(t_2, c_2))∈Fcons	Foreign key of table t_1 points to primary key of table t_2.
sub(t_1, t_2)∈ SInh	sub() represents a single inheritance relationship and table t_1 is a child of table t_2
subM(t_1, t_2, ….)∈ MInh	subM() represents a multi-inheritance relationship and table t_1 is a child of multi tables.
parent(t)∈Tab	parent(t) represents the set of all the parent tables of table t.

4.1 The Overall of Storage Framework

RDF Schema represents information about classes and properties. Each table in an object-relational database corresponds to a set of entities in the real world and ORDBs support the inheritance relationship between tables. So, the structure of RDF Schema is very similar to the structure of ORDBs. In this section, we propose an object-relational storage model by analyzing the structure information of RDF Schema.

The main idea of the storage framework proposed in this paper is to respectively create a table for each class and each property in RDF Schema. We then use the created table to store the corresponding instances of the class (or property). In many real applications, the number of class and property are often much smaller than instances. Actually, our storage framework based on ORDBs mainly contains five types of tables: *class table, property table, property-relation table, property-type table, and namespace table*.

For the RDF(S) shown in Figure 2, for example, Figure 3 shows the object-relational storage model generated by the RDF Schema in Figure 2. We take Figure 3 as an example to illustrate the overall architecture of RDF(S) storage with the ORDBs. In Figure 3, there are totally four class tables (i.e., *university, school, student,* and *person*), two property tables (i.e., *degreefrom* and *doctordegreefrom*), two property-type tables (i.e., *pro_domain* and *pro_range*), one property-relation table (i.e., *property_relation*), and one namespace table (i.e., *bief_namespace*).

According to the formal definitions proposed in Section 2, $RM = (RB, RA, RI)$ is used to represent RDF(S) model and $ORDBM = (Basic, Cons, Inh, Ins)$ is used to represent object-relational database model. We use ρ to express the process of creating tables.

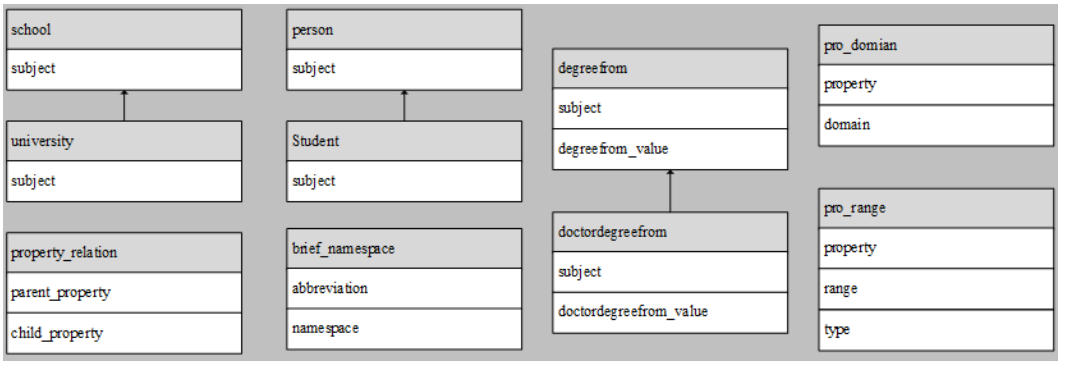
1. ProTable(property table): $\forall$ p$\in$ RP THEN ρ (p) $\in$ ProTable

We can create a table for each property in RDF Schema and the table name is the property name. The table contains two fields. The first field is named *subject* which is used to store instance's subject. The second field is named the property name and is used to store instance's object. In Figure 3, the *degreefrom* table and *doctordegreefrom* table are created by the *degreeFrom* property and *doctorDegreeFrom* property in RDF Schema, respectively.

2. ClaTable(class table): $\forall$ c$\in$ RC THEN ρ (c) $\in$ ClaTable AND subM(ρ (c), ρ (pro(c)), ρ (parent(c)))

We also create a table for each class in RDF Schema and the table name is the class name. This table contains only one field named *subject*, which is used to store all instances of this class. When a

Figure 3. Object-relational storage model



class table is created, it should inherit from its parent class table, along with property tables created by all properties whose domain is this class. In Figure 2, *Person* class is the parent of the class *Student* and the domain of the *doctorDegreeFrom* property is the class *Student*. The *student* table inherits from the *Person* table and the *doctordegreefrom* property table. So, the *student* table owns all the fields in the parent table and the property table.

3. ProRelTable(property-relation table): ∀ spo(p_1,p_2) THEN ρ (spo(p_1, p_2))∈ProRelTable

Property inheritance relationships are not treated in the same way as class tables. Instead, the *property-relation* table is created to store inheritance relationships between properties. *Property-relation* table contains two fields, *child_property* and *parent_property* like the *property_relation* table in Figure 3.

4. ProTypeTable(property-type table): ∀ p∈ RP THEN ρ (dom(p)) ∈ProTypeTable AND ρ (ran(p)) ∈ProTypeTable

ProTypetable is used to store property's domain and range. In Figure 3, *pro_domain* table contains two fields, *property* and *domain*, to store all the defined domains of the property. *Pro_type* table contains three fields: *property*, *range*, and *type*, which are used to store name, range and type of the property.

5. NameSpaTable(namespace table): ∀ name(t) THEN ρ (name(t))∈NameSpaTable

To improve the readability of RDF(S), RDF(S) defines abbreviations to describe commonly used namespaces. A large number of resources in RDF(S) shares the same namespace. Duplicate storage can cause a large amount of wasted space. So, we create a namespace table to store the namespaces defined by RDF(S) and their corresponding abbreviations like the *brief_namespace* table shown in Figure 3.

4.2 Storage Rules

The previous section describes the overall framework of RDF(S) store with ORDBs. The framework mainly preserves the inheritance relationships of classes in RDF Schema. Based on the framework, this section explains the storage rules from two aspects: storing RDF Schema and storing instances. This section also adopts *RM* = (*RB*, *RA*, *RI*) as an RDF (S) model and *ORDBM* = (*Basic*, *Cons*, *Inh*, *Ins*) as an object-relational database model. The symbols ψ represents the storing procedure. First, let's introduce the storage of RDF Schema.

Rule 1 (storing namespace): ∀*i*∈ IRI THEN ψ(name(*i*)) → NameSpaTable

All namespaces defined in RDF(S) can be stored into the namespace table. This table contains two fields. The *abbreviation* field stores the abbreviation of the namespace and the *namespace* field stores the full namespace. Figure 4 shows an example of storing RDF(S) namespace.

Rule 2 (storing relationships of properties): ∀spo (p1, p2) THEN ψ(p1, p2) → proRelTable

The inheritance relationship between different properties can be stored in the *property_relation* table. Figure 5 shows an example of storing inheritance relationship between properties. In Figure 2, *degreeFrom* is the parent property of the *doctorDegreeFrom* property. This relationship can be stored directly in the *property_relation* table as shown in Figure 5.

Rule 3 (storing property constraint): ∀p∈ RP THEN ψ(dom(p)) ∈ ProTypeTable AND ψ(ran(p)) ∈ProTypeTable

Figure 4. An example of namespace storage

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/22-rdf-syntax-ns#"
  xmlns:rdf="http://www.w3.org/rdf-schema#"
  xmlns:rdf="http://www.example.com/rdf/#">
```

brief_namespace

abbreviation	namespace
rdf	http://www.w3.org/22-rdf-syntax-ns#
rdfs	http://www.w3.org/rdf-schema#
ub	http://www.example.com/rdf/#

Figure 5. An example of property relationship storage

```
<owl:ObjectProperty rdf:about=
"http:// www.example.com/rdf/#doctoralDegreeFrom">
<rdf:subPropertyOf rdf:resource=
"http:// www.example.com/rdf/#degreeFrom"/>
< /owl:ObjectProperty >
```

property_relation

child_property	parent_property
ub:doctoralDegreeFrom	ub:degreeFrom

To preserve the full semantics of property, the domain, range, and the type of the property need to be stored in the *property -type* table. Figure 6 shows an example of storing constraint information related to a property.

In Figure 6, there is an object property *doctoraldegreeFrom* and a datatype property *name*. *pro_domain* table and *pro_range* table are used to store the domain, range and property type of *doctoralDegreeFrom* and name. rdf: ID =" doctoralDegreeFrom" is the abbreviation of rdf: about = http://www.example.com/rdf/#doctoralDegreeFrom doctoralDegreeFrom".

Rules 1-3 store the RDF Schema semantic that is not retained by the object-relational framework created in Section 4.1. Then we introduce rules about storing RDF instances.

Rule 4 (storing instances of class): ∀type ((sub(t))∈RC THEN ψ(sub(t)) → claTable (type(sub(t)))

Each instance has a corresponding class. Figure 7 shows how the class instance is stored. In this example, *University0* and *Univeristy1* are both instances of the class *University*. So, the resource is stored in the corresponding class table. As the analysis in Section 4.1, each class table has a subject field that stores all instance resources.

Rule 5 (storing property resources whose domain is not empty): ∀pre(t)∈RP AND dom(pre(t)) ∈ RC THEN ψ(t) → ClaTable(dom(pre(t)))

The domain of a property restricts the subject type of the property instance. Usually, the domain of a property is a class. If the domain of the property is not empty, the class table has the field of the current property according to Rule 2 in Section 4.1. Property instances whose domain is not empty can be stored directly in the corresponding class table. For example, the domain of *name* and *telephone* property in Figure 8 are both *Student* class. So, the *student* table inherits the *name* property table and

Figure 6. An example of property constraint storage

```
<owl:ObjectProperty
rdf:ID="doctoralDegreeFrom">
<rdfs:domain rdf:resource="#Student"/>
<rdfs:range rdf:resource="#University"/>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="name">
<rdfs:domain rdf:resource="#Person"/>
<rdfs:range rdf:resource="#xsd:String"/>
</owl:DatatypeProperty>\
```

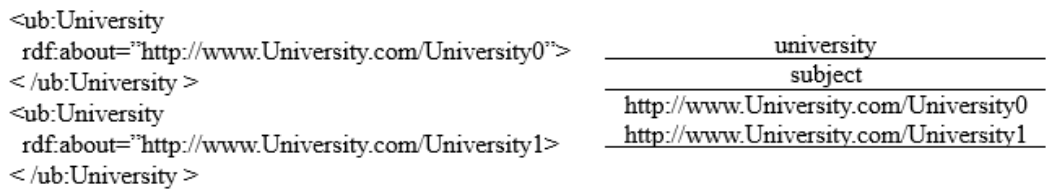
pro_domain

property	domain
ub:doctoralDegreeFrom	ub:student
ub:name	ub:person

pro_range

property	range	type
ub:doctoralDegreeFrom	ub:university	objectProperty
ub:name	xsd:string	datatypeProperty

Figure 7. An example of class instance storage



```
<ub:University
  rdf:about="http://www.University.com/University0">
< /ub:University >
<ub:University
  rdf:about="http://www.University.com/University1>
< /ub:University >
```

university
subject
http://www.University.com/University0
http://www.University.com/University1

telephone property table and have *name_value* field and *telephone_value* field. The instance *Mike* in Figure 8 can be directly stored in the *student* table.

There are multi-valued properties in RDF(S). A row of data in a class table cannot store multiple values of the same property. Therefore, if the property value is found to already exist when you insert it, the property is a multi-valued property. Then the new value should be stored in the property table not in the class table. This method can effectively solve the problem of multi-valued properties. It is important to note that when querying a multi-valued property, the property table needs to be queried. This is because the property table contains all information about the current multi-valued property, while the class table contains incomplete data.

Rule 6 (storing property resources whose domain is empty): $\forall pre(t) \in RP$ AND $dom(pre(t)) = \varnothing$ THEN $\psi(t) \rightarrow proTable(pre(t))$

If the domain of a property is empty, this property can belong to any class. So, according to the framework created in Section 4.1, there is not a class table that inherits this property table. Figure 9 shows an example of storing property resources whose domain is empty. In this example, the domain of the property *ID* is empty. So, the instance of the property is stored directly in the property table rather than the class table. This may result in joins between tables when querying data, but it avoids a large number of null values in class tables.

4.3 Storage Algorithm

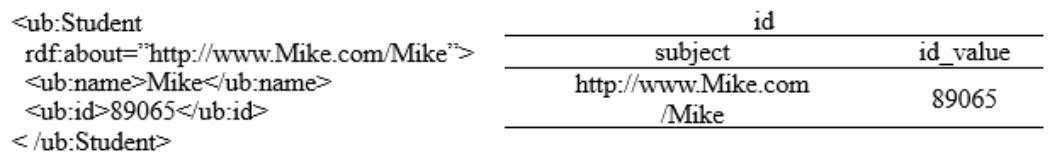
This paper designs two data structures, RDFClass and RDFProperties, to store the structural information analyzed from RDF Schema files. RDFClass stores the *class name* of the current class, the direct parent class, and a collection of attributes that define the domain as that class. RDFProperties

Figure 8. An example of storing the property with no-empty domain

```
<ub:Student
  rdf:about="http://www.University0.com/Mike">
  <ub:name>Mike</ub:name>
  <ub:telephone>876-567-35</ub:telephone>
< /ub:Student>
```

student		
subject	name_value	telephone_value
http://www.University0 .com/Mike	Mike	876-567-35

Figure 9. An example of storing the property instances with empty domain



```
<ub:Student
  rdf:about="http://www.Mike.com/Mike">
  <ub:name>Mike</ub:name>
  <ub:id>89065</ub:id>
< /ub:Student>
```

id	
subject	id_value
http://www.Mike.com /Mike	89065

store the *property name* of the current property, the direct parent class of the property, as well as the definition domain, value domain, and property type of the property.

The creation of an object-relational storage model for RDF(S) is described in Table 3. The input of this algorithm is the RDF(S) and the JDBC connection (connecting to the database). The output is the created object-relational storage model, *classes* set and *properties* set. Among them, each element in the classes set is an instance of RDFClass, and each element in the properties set is an instance of RDFProperty.

The detailed process of creating an object relational storage model based on RDF Schema is as follows:

1) Firstly, create a *namespace table*, *property type table*, and *property relationship table* in the database (steps 1-3). These three tables are directly created based on the storage structure designed in section 4.1, and the namespaces and their abbreviations defined in the RDF Schema are stored in the namespace table (step 4).
2) Secondly, read the RDF Schema file, extract all properties from the RDF Schema file, store the *definition domain*, *value domain*, and *property type* of the attributes to the RDFProperties, and save all RDFProperties instances to the *properties* set (step 5). Then extract all the classes in the RDF Schema, store the *class name*, *direct parent class*, and other information to the RDFClass, and store all RDFClass instances to the *classes* set (step 7).
3) Then, traverse each instance in the *properties* set, create a *property table* containing *subject* and *value* fields based on the name of the property (step 9), store the inheritance relationships contained in the property in the created *property relationship table* (step 10), and then save the definition domain, value domain, and data type of the property in the property type table (step 11). Due to the fact that each RDFClass structure also defines a set for storing properties of the current class in the domain, while traversing the property, the name of the attribute needs to be added to the *classes* set represented by the domain (step 12). The purpose of doing this is to facilitate the creation of a *class table* in the future, where all the properties contained in the current class can be directly obtained through the *classes* set.

Table 3. Algorithm of for creating an object-relational storage model

Input: RDF(S), Database connection (c) **Output: classes set, properties set, and object-relational storage model**

1. createNamespaceTable(c);
2. createPropertyReltiaonTable(c);
3. createPropertyConstrainTable(c);
4. *namespaceTable*←extractNamespace(rdfSchemaFile)
5. properties←extractProperty (rdfSchemaFile); //Extract all properties from the RDF Schmea
6. *classes*←extractClass(rdfSchemaFile); //Extract all classes from the RDF Schmea
7. For *i* in *properties* do
8. createPropertyTable(c) //Create a *property table* for each property
9. insertPropertyRelation(c*)* //Store the inheritance relations of property in the *property-relation table*
10. insertPropertyType(c) //Store the type relationship of each property in the *property-type table*
11. addClassDomain(classes, *i*); //Add the name of an property to the *classes* set represented by the definition domain
12. End For
13. For *i* in *classes* do
14. *stack*←findParentClass(*i*)
15. If *stack! = null* then
16. createParentClass(stack,c, classes);
17. Else
18. createClassTable(classes, c)//Created *class table*
19. End if
20. End For

4) Finally, traverse each RDFClass instance in the *classes* set. Each instance contains the name information of the current *class*, *direct parent class* information, and all *properties* of the defined domain for that class. Process the parent class relationship to determine whether the data table corresponding to the *direct parent class* of the current class has been created. If it has already been created, create a *class table* directly based on the current class and inherit the data table where the parent class is located and all properties are located. There is no need to repeat inheritance for properties already owned in the parent class table (step 19). If it has not been created, then the parent class of the parent class needs to be searched along the inheritance chain, and all classes traversed along the way are pushed into a stack structure named *stack* until the parent class of the already created data table or the top-level parent class of the not created data table is found (Step 15). Then create a data table for each class in the stack in the order of stack exit (step 17).

In this algorithm (as shown in Table 3), there are two *For* loops. The time complexity of the first *For* loop is O(*m*), where *m* represents the number of all properties in the properties set. The time complexity of the second *For* loop is O(*n*), where *n* represents the number of all classes in the classes set. Therefore, the overall time complexity of the algorithm is O(m+n).

With the created object-relational storage model for RDF(S), the RDF(S) file can be stored by applying the mapping rules above. The storage of RDF instances in the ORDBs is described in Table 4. The input of this algorithm is the RDF file and *classes* set obtained from the algorithm above, and the output is the data tables that store RDF data.

The detailed process of storing RDF instance data into an object relational database is as follows:

1) Firstly, extract all instance data from the RDF file (Step 1), and store the *namespaces* defined in the RDF file in the *nam*espace table (Step 2).
2) Secondly, obtain each instance in the RDF file in sequence and analyze the type of the current instance (Step 4). Then create a HashMap set called propertyValue. The key is used to store the properties of the current instance, and the value is used to store the corresponding property values (Step 5).
3) Then, traverse all properties in the current instance in sequence. If the definition domain of the current property does not include the class to which the instance belongs, it means that

Table 4. Algorithm for storing RDF instance data

Input: RDF file, Database connection (c), *classes* **set** **Output: Data tables for the RDF data**

1. *instances*←extractInstance (rdfFile); //Extract instance data from RDF file
2. *namespaceTable*←extractNamespace(rdfFile)
3. while *instances.hasNext* do
4. *type*←getRDFtype(i) //Get the class that the current instance belongs to
5. *propertyValue*←createHashMap() //Create a HashMap structure to save property values
6. For *j* in *Allpre(i)* do //Traverse all properties of the same subject resource
7. if *j*∈*classes(type).getdomain* then //The definition domain of the property
8. If *propertyValue.contains(j)* then //Multivalued property
9. insertintoProperty(sub(*j*), obj(*j*))
10. Else propertyValue.add(j, obj(*j*))
11. End if
12. Else
13. insertintoProperty(sub(*j*), obj(*j*))
14. End if
15. End For
16. InsertintoClass(*i*, propertyValue) //Insert into *class table*
17. End while

the property cannot be inserted into the class table. Then, it can be directly inserted into the property table (skip to step 13). Otherwise, it is necessary to determine whether a multivalued property appears. If the propertyValue set already contains the current property, it indicates that a multivalued property has appeared. Then, the multivalued attribute is directly inserted into the property table (steps 8-9). Otherwise, place the property and corresponding property values into the propertyValue set (step 10).

4) Finally, after the *For* loop ends, insert the properties and property values from the obtained propertyValue into the *class table*. The function of propertyValue is to use memory to store multiple properties and property values of the same instance, to insert a row of data in the *class table* once, avoiding multiple update operations on the *class table* and improving storage efficiency.

In this algorithm (as shown in Table 4), a *while* loop is included, mainly used to traverse all instances in the RDF file. A *for* loop is nested within the *While* loop to traverse all the properties contained in the current instance. Therefore, the overall time complexity of the algorithm is $O(m \times \text{Max}(n))$. Where m represents the number of instances in the RDF file, and max (n) represents the maximum number of properties contained in the instance.

4.4 Query Method

At present, SPARQL language is the officially recommended query language for querying RDF(S), while the common query language for databases is SQL language. The core idea of SPARQL query statements is graph-based queries. The graph patterns in SPARQL can be divided into *basic graph pattern*, *composite graph pattern*, *optional graph pattern, multi graph pattern*, and *value constrained graph pattern*.

1) The *basic graph pattern* is the foundation of all graph patterns. Its structure is the same as the triple pattern, consisting of three parts: *subject*, *predicate*, and *object*. For example, ?X eo:name ?Y is a basic graph pattern. ? identify the variable, where eo:name is a constant. The semantics of this statement are to select the *subject* and *object* of a triple instance with the predicate name.
2) The *combination graph pattern* is a combination of multiple basic graph patterns, and the returned query results need to meet each basic graph pattern. The idea of the combination graph pattern can correspond to the inner join operation in SQL statements.
3) The *optional graph pattern* utilizes the OPTIONAL keyword to connect different graph patterns, and the returned query results may not meet the graph pattern modified by the OPTIONAL keyword. The idea of optional graph patterns can correspond to left join operations in SQL statements.
4) The *multi graph pattern* utilizes the UNION keyword to connect different graph patterns, and the returned result set should contain the query results for each graph pattern. The idea of this pattern can also correspond to the UNION keyword in SQL statements.
5) The *value constrained graph pattern* mainly constrains the query results, and keywords such as *Limit* and *Order* by can be selected to modify the result set. The keywords provided by SPARQL also have corresponding keywords that can be directly converted in SQL statements.

In summary, the key to converting SPARQL statements into SQL statements is how to map the graph patterns of SPARQL. In (Chebotko *et al*., 2009), an algorithm called BGPtoSQL was proposed to convert SPARQL statements of the graph patterns into equivalent SQL statements. SQL statements can mainly be decomposed into patterns such as SELECT, FROM, and WHERE. The SELECT keyword is followed by the fields that need to be displayed as query results, the FROM keyword is followed by the data table that needs to be queried, and the WHERE keyword is the query condition that needs to be met. The BGPtoSQL algorithm models the graph pattern in SPARQL query statements as a

directed graph BGP=(N, E), where N is the node set representing the *subject* and *object*. E is the edge set used to represent the *predicate*. Each edge points from the *subject* node to the *object* node. Firstly, add the data table corresponding to each edge in BGP to the FROM clause, and add the variables in the statement to the SELECT clause to construct the query result set. Then, construct corresponding WHERE query conditions by determining whether each node is a variable or a constant, as well as the degree of entry and exit of the node. Sequentially process each basic graph pattern, perform join operations on the basic graph, and parse SPARQL query statements into SQL queries.

Based on the BGPtoSQL algorithm, this study further adjusts the converted SQL statements to adapt to the storage structure proposed in this paper. The data table obtained through the BGPtoSQL algorithm includes a class table and a property table. Since the class table inherits the fields in the property table, if the class table in the FROM clause is a child table of the property table to be queried, the property table is removed from the FROM clause and the query conditions for the property table in the WHERE clause is converted into queries for the same fields in the class table. If the queried property is a multivalued property, then the property table can be directly queried, and the property values of the field cannot be obtained through the class table. This is because for multivalued properties, only the data stored in the attribute table is complete.

The RDF (S) storage model proposed in this paper includes both property type tables and property relationship tables that store property semantics, and different class tables also express inheritance semantics. Therefore, the resulting SQL statements can be repackaged to achieve inference queries on the stored RDF(S). When conducting inference queries, the main focus is on analyzing the data tables queried after from.

When the queried data table is a class table, the ORDB supports querying all data in the subclass table through the parent class table. Therefore, when querying the parent class table, all data in the subclass table can be directly obtained without the need for additional SQL statements.

When the queried data table is a property table, it is necessary to use the property relationship table to analyze the inheritance relationship of properties, obtain all sub properties of the current property through the property relationship table, and query each sub property table. Figure 10 shows an example of using property relationship table for inference queries. When the data table to be queried is a *degreefrom* data table, the table contains two sub properties in the property relationship table, namely *doctoraldegreefrom* and *masterdegreefrom*. Therefore, the results of the inference query should also include data that meets the conditions in both *doctoraldegreefrom* and *masterdegreefrom* data tables. Use the UNION keyword to merge query results that meet the same conditions in the sub property table.

5. EXPERIMENT AND EVALUATION

Based on the storage framework and rules proposed in Section 4, we implemented a prototype system named RDF2ORDB, which can store RDF(S) into the ORDBs. In this section, we present the prototype system and validate the system with experiments.

5.1 Implementation

Experimental environment: In this section, we briefly discuss the implementation of the RDF2ORDB. The prototype system is run on a PC with Intel Core i5, 2.50GHz CPU, 8G RAM and Windows 10 operating system. The programming language is JAVA and the version is JDK1.80. The development tool is IDEA. The database used is PostgreSQL and the version is 10.12.

The prototype consists of three main modules: *parsing module*, *storage module* and *query module*. The overall architecture of the RDF2ORDB is shown in Figure 11.

Figure 10. An instance of converting SPARQL query statements into SQL query statements

The original SPARQL statement:

```
SELECT ?X
WHERE
{?X ub:degreeFrom <http://www.Department0.University0.edu>}
```

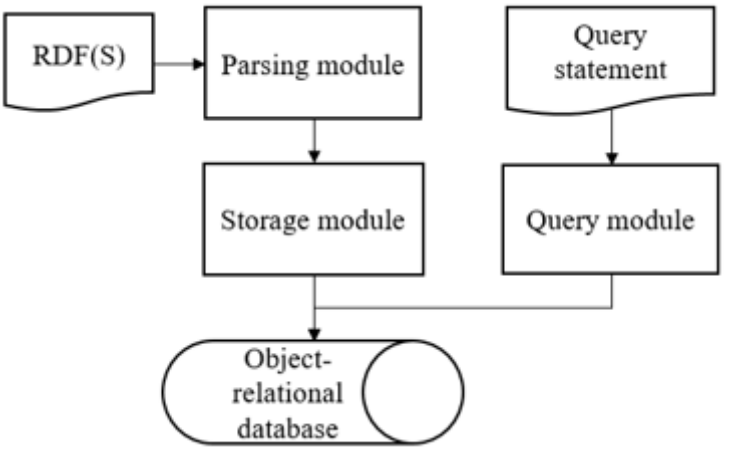
The converted SQL statement:

```
SELECT degreefrom.subject
FROM degreefrom
WHERE  degreefrom.degreefrom_value=<http://www.Department0.University0.edu>
```

Inference query SQL statement:

```
SELECT subject FROM degreefrom
WHERE degreefrom_value=<http://www.Department0.University0.edu>
UNION
SELECT subject FROM doctoraldegreefrom
WHERE  doctoraldegreefrom_value=<http://www.Department0.University0.edu>
UNION
SELECT subject FROM masterdegreefrom
WHERE  masterdegreefrom_value=<http://www.Department0.University0.edu>
```

Figure 11. The overall architecture of the RDF2ORDB



1) Parsing module: this module receives the RDF(S) file uploaded by users and then parses classes, properties and properties contained in the file. This module finally creates the data tables: *class table*, *property table*, *property-relation table*, *property-type table* and *namespace table*, which are included in the object-relational storage model. The creation of an object-relational storage model for RDF(S) is described in Table 3.
2) Storage module: this module stores the parsed results of RDF(S) file in the data tables. The storage of RDF instances in the ORDBs is described in Table 4.

3) Query module: this module mainly generates the corresponding SQL statement from the SPARQL statements issues by users. We follow the mapping of SPARQL-to- SQL developed in (Rodriguez-Muro and Rezk, 2015).

The screen snapshot RDF2ORDB running one of the case studies is shown in Figure 12. At the top of the interface, users can upload the RDF Schema and RDF files that need to be stored in the ORDBs. After successful storage, the left side of the interface displays the created table and fields it contains. A query interface is provided on the right side of the system, where users can directly enter their SPARQL statements. In the figure, users would retrieve the faculties of assistant of professor in the given university. The query results are shown at the bottom of the screen.

5.2 Experimental Dataset

We adopted LUBM dataset (Lehigh University Benchmark) developed in (Guo, Pan and Heflin, 2005). LUBM provides an ontology built around the university domain, containing 43 classes and 32 attributes (25 object properties and 7 datatype properties), which describes the complex relationships among various kinds of departments and professors within the university. LUBM provides a UBA[9] (Univ-Bench Artificial) data generator tool that can extend RDF instances of any size based on ontology. LUBM was chosen as the experimental dataset because the domain described by LUBM is familiar to most users, and there are a moderate number of classes, each of which contains complex semantic relations. Moreover, LUBM is widely used in various benchmarks and is the authoritative benchmark set at present. In this paper, two test datasets, dataset1 and dataset2, were built by using LUBM dataset. The detailed information about test datasets is shown in Table 5.

Figure 12. The screen snapshot of RDF2ORDB

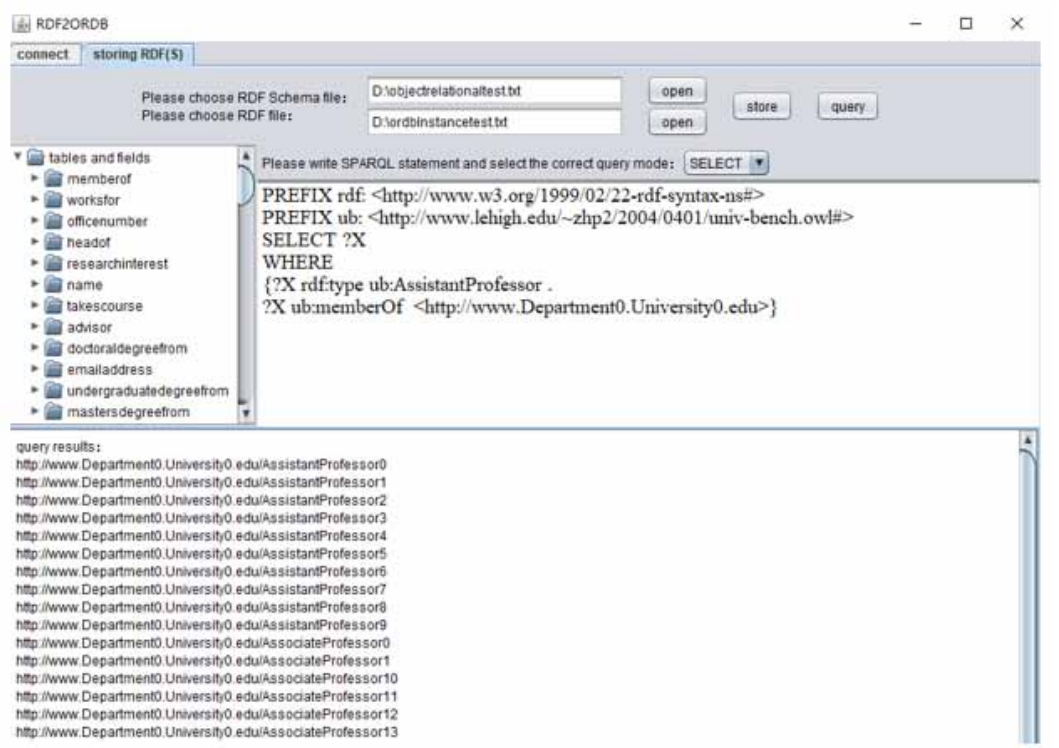


Table 5. Experimental datasets

Name	Dataset	Number of RDF Triples	Size of Dataset (M)
dataset1	LUBM (1, 0)	103074	12.6
dataset2	LUBM (8, 0)	1035608	132.8

Data generation. We generate data with the UBA, a tool that is developed for the benchmark. This data generator has the capability of generating data randomly and repeatably. The minimum unit of generating data, for example, is a university. Then for each university, it contains a set of OWL files describing its department and student information (as shown in Figure 13), where instances of classes and attributes are randomly determined. In addition, to make the generated data as realistic as possible, some conditions for generating data are set based on common sense and domain knowledge. Let us look at an example. We can set "the range of the number of departments in a university is set to [10,25]", "each student should take at least one course or at most three courses", and so on. The data generator then identifies universities by assigning different indexes to them, for example, naming the first university as University0, and etc. Also, through the data generator, it is possible to manually set how many universities and which ones to generate. Finally, based on the OWL files created by the generator, we count the number of RDF triples contained in the OWL files, and this provide a data foundation for subsequent query and inference tasks.

5.3 Analysis of Experiment Results

5.3.1 Analysis of Storing and Querying Performance

The existing method of storing RDF(S) based on the ORDBs is still immature. Tools such as RDF Suite and Sesame support storing RDF(S) using an object-relational schema, but their query efficiency is not as good as relational databases. To evaluate the performance of the storage scheme proposed in this paper more objectively, we choose the relatively mature storage mode of the relational database for comparison. The effectiveness of the storage method proposed in this paper is compared with three representative schemes: *Triple* (vertical storage), *SW-store* (horizontal storage) and *Jena* (type storage), which adopt different relational structures. We do not compare our method with few proposals that store RDF(S) with OODBS and ORDBs because they are not experimentally evaluated at all. In addition, different storage structures have great impact on the query efficiency. Therefore, the analysis of performance is evaluated from two aspects: storage performance and query performance.

Figure 13. A simple example of generating data

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:rdfs="http://www.w3.org/2000/01/rdf-
schema#" xmlns:owl="http://www.w3.org/2002/07/owl#" xmlns:ub="http://www.lehigh.edu/~zhp2/2004/0401/univ-
bench.owl#">
  ------
  <ub:UndergraduateStudent rdf:about="http://www.Department0.University0.edu/UndergraduateStudent460">
    <ub:name>UndergraduateStudent460</ub:name>
    <ub:memberOf rdf:resource="http://www.Department0.University0.edu"/>
    <ub:emailAddress>UndergraduateStudent460@Department0.University0.edu</ub:emailAddress>
    <ub:telephone>xxx-xxx-xxxx</ub:telephone>
    <ub:takesCourse rdf:resource="http://www.Department0.University0.edu/Course18"/>
    <ub:takesCourse rdf:resource="http://www.Department0.University0.edu/Course24"/>
    <ub:takesCourse rdf:resource="http://www.Department0.University0.edu/Course42"/>
  </ub:UndergraduateStudent>
  ------
</rdf:RDF>
```

(1) Analysis of storage performance

We evaluate storage performance from two aspects: storage time and storage size.

Storage time. In this paper, we use datasets with different size to obtain the storage time of four storage methods by. Each storage method conducted 6 experiments for different datasets, of which the first experiment was used as a preheating system and the experimental results were not recorded. The final storage time was obtained by calculating the average storage time of the last five times. Figure 14 shows the storage time of the four storage methods when two test datasets are stored.

It is shown in Figure 14 that our method takes the longest time in RDF data store because it needs to parse RDF(S) first, but its storage time is roughly comparable to the storage time of Triple and SW-store. It is also shown in Figure 14 that Jena has the shortest storage time among these four methods. This is because Jena is a memory-based framework, which directly stores RDF triples without a complex parsing process. However, such a simple processing disadvantages Jena in other performance aspects.

Storage size. To further validate the effectiveness of our RDF (S) storage method, after the prototype system loads the test datasets, we analyze the size of the resulting repository. The size, which is tested only for systems with persistent storage, is calculated based on the total size of all files that make up the repository. For two datasets with different sizes, Triple, SW-Store, Jena and our RDF2ORDB are used to store the test data. After loading the data, the size of each repository is shown in Figure 15.

It is shown in Figure 15 that SW-store occupies the largest storage space and our approach occupies the smallest storage space. The major reasons are that: (a) for many duplicate prefix IRIs in RDF(S) data, our approach uses the namespace table to store unique prefixes so that other tables only need to store the ID of the prefix in the namespace table, avoiding the waste of storage space caused by duplicate prefix IRIs; (b) there are a large number of duplicate subjects, predicates, and objects in the stores of Triple and SW-store, however, our method uses subject table, predicate table and object table, and only needs to store unique data, avoiding the duplicate storage. It is also shown in Figure 15 that, among three comparison methods, Jena occupies the smallest storage space, but it is very larger than our approach and it is especially true for large-scale datasets (e.g., dataset2).

(2) Analysis of query performance

The storage structure directly affects the retention of RDF(S) semantics and the efficiency of queries. We verified through queries if our storage method can fully preserve RDF(S) semantics and efficiently return query results. We used LUBM dataset to evaluate the query efficiency of

Figure 14. The storage time of datasets

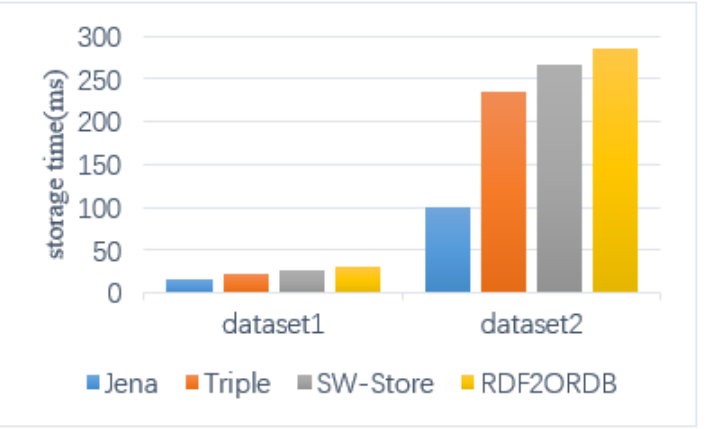
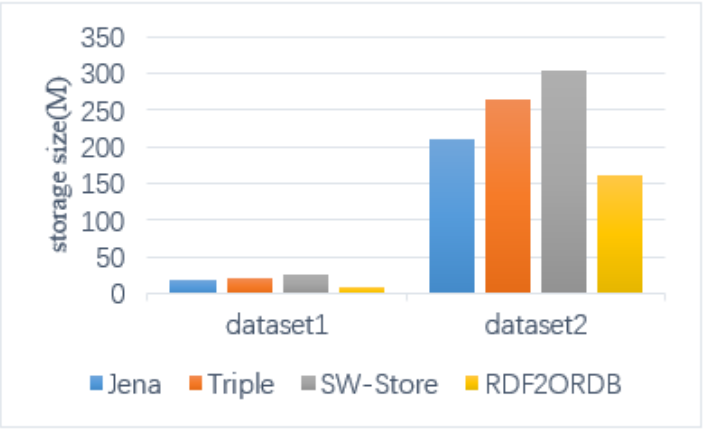


Figure 15. The storage size of datasets



RDF2ORDB, with two main indicators: *query soundness* (measured by analyzing how many correct answers are returned) and *completeness* (measured by analyzing how many returned answers are correct). Specifically, LUBM provides 14 predefined test queries, in which several test queries have a simple structure and strong similarity. So, based on the structural characteristics of predefined test queries and the SPARQL[10] query structure type, we only selected 8 test queries (Q1-Q8) from the LUBM test query set. Here SPARQL is the standard query language recommended by W3C for RDF. A basic SPARQL query consists of a SELECT clause, which is followed by query variables represented by bound variables (variable with specified value) that appear in the result set, and a WHERE clause, which is followed by graph patterns that match against the RDF graph that the query is being executed on.

These 8 selected test queries are explained as follows.

(a) Q1, Q2 and Q3 are simple selective queries (single tuple query), which directly obtain the required data by querying the corresponding data table. This linear shaped pattern consists of a set of triple patterns, where the subject and object are connected by different unique connecting variables. That is, the connecting variables are located at the subject position in one triple pattern and at the object position in another triple pattern.
(b) Q4 is a star shape query with high selectivity (star query), in which the query statement contains multiple attribute information. The star shaped pattern consists of a set of triple patterns, connected together by a single connected variable of the subject position or object position.
(c) Q5 and Q6 are snowflake shape queries (chain query), where the object of the previous query triplet is the subject of the next query triplet. The snowflake shaped pattern consists of several star shapes connected by different connecting variables at the position of the subject or object in a triple pattern.
(d) Q7 and Q8 are hybrid queries that combine the characteristics of star query and chain query (mixed query), which contain a large number of intermediate results. This complex structure is a combination of the above query patterns.

In summary, the above 8 selected test queries cover four different types of query structures in SPARQL (i.e., linear shape pattern, star shape pattern, snowflake shape pattern, and complex structure). which can effectively evaluate the impact of different query types. To comprehensively verify the

query efficiency of our storage method, we used the 8 test queries Q1-Q8 to effectively evaluate the query efficiency of RDF2ORDB. These 8 selected test queries are presented in Figure 16:

Table 6 shows the query response time for the dataset1. Query time depends not only on the size of the dataset, but also on the complexity of the query statement. In the query of Q1-Q3 statements, the four methods have achieved good results. This kind of query form is simple. Using the method of this paper can directly query the corresponding table to get the required data.

Q4 is a star query. For this type of SPARQL, RDF2ORDB works best. Because the class table contains all the property fields whose domain is this class, which is suitable for star query. Triple storage scheme is the worst. The triple storage requires a large number of self-joins when querying, which has a great impact on efficiency.

Q5-Q6 is a chain query, where the object of the previous query statement is the subject of the next query statement. For this kind of query, RDF2ORDB is the best. This is also because the class table contains property fields, so only a few joins between tables are needed when querying. The hybrid query of Q7-Q8 combines the characteristics of star query and chain query, and RDF2ORDB still achieves good performance.

Viewed from query performance in Section 5.3.1, it is shown in Table 6 that Jena stores RDF data in memory and traverses the entire dataset for querying. Its query time for different SPARQL

Figure 16. SPARQL query statements

Single tuple query	Star query

Query1 (Q1): SELECT ?X WHERE {
?X ub:publicationAuthor
http://www.deparment0.university0.edu/AssistantProfessor0}
Query2 (Q2): SELECT ?X WHERE {
?X ub:takeCourse
http://www.deparment0.university0.edu/GraduateCourse0}
Query3 (Q3): SELECT ?X WHERE {
?X rdf:type ub:UndergraduateStudent}
	Query4 (Q4): SELECT ?X ?Y1 ?Y2 WHERE {
	?X rdf:type ub:FullProfessor .
	?X ub:workFor
	http://www.deparment0.university0.edu .
	?X ub:emailAddress ?Y1 .
	?X ub:doctoralDegreeFrom ?Y2}

Chain query	**Mixed query**

Query5 (Q5): SELECT ?X ?Y WHERE {
?X rdf:type ub:UndergraduateStudent .
?Y rdf:type ub:AssistantProfessor .
?Z rdf:type ub:Course .
?X ub:advisor ?Y
?Y ub:teacherOf ?Z}
Query6 (Q6): SELECT ?X ?Y WHERE {
?X rdf:type ub:FullProfessor .
?Y rdf:type ub:Department .
?X ub:workFor ?Y
?Y ub:subOrganizationOf http://www.university0.edu}
	Query7 (Q7): 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}
	Query8 (Q8): SELECT ?X ?Y ?Z WHERE {
	?X rdf:type ub: UndergraduateStudent .
	?Y rdf:type ub:Department .
	?X ub:memberOf ?Y .
	?Y ub:subOrganizationOf
	http://www.university0.edu .
	?X ub:emailAddress ?Z}

Table 6. Query response time of dataset1 (ms)

Method	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Triple	78	60	50	563	25841	10066	28494	29825
SW-Store	46	52	60	62	13869	2521	751	3976
Jena	106	102	107	97	119	105	107	93
RDF2ORDB	**38**	**39**	**42**	**46**	**56**	**34**	**32**	**59**

queries is relatively stable. While Triple and SW-Store have better processing ability for simple queries (Q1-Q4), their query efficiencies are not satisfactory for complex queries (Q5-Q8) because a large number of self-join and join between tables are needed. Our approach obtains properties of class by querying the class table and this avoids a large number of null values. On the small-scale dataset (i.e., dataset1), our RDF2ORDB achieves the shortest query response time for all eight queries. For the large-scale dataset (i.e., dataset2), it is shown in Table 7 that, due to so many inner-join and out-table join operations, Triples and SW-store take too long to obtain results within a reasonable interval for complex SPARQL queries. "NaN" in Table7 indicates that the query processes were interrupted because they took too long. It is also shown in Table 7 that the query time of Jena for different SPARQL queries is still relatively stable. But the query time of our approach is better than Jena for most queries expect for Q3.

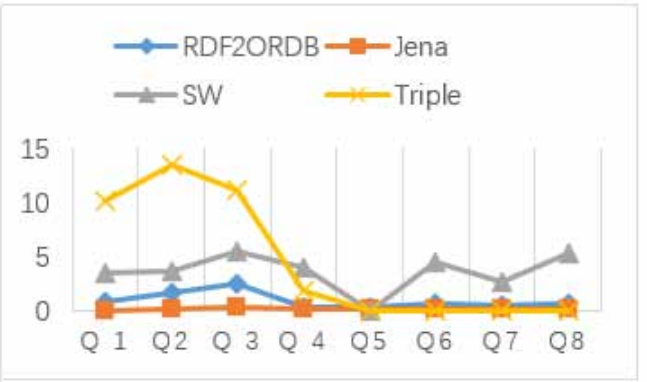
To more intuitively reflect the change of the query time with the increase of data size, Figure 17 shows the increase ratio of the query time. In Figure 17, the abscissa represents different query statements and the ordinate represents the value of (t2-t1)/ t1, where t1 is the query time for dataset1 and t2 is the query time for dataset2.

It is shown in Figure 17 that triple storage and SW-store are greatly affected by the increase in data size. However, the query time of Jena and RDF2ORDB increases steadily. When querying with Q2 and Q3, the query time increment ratio of RDF2ORDB is high. This is because Q2 and Q3 are single tuple queries for a certain class with low selectivity. With the increase of data size, the number of instances of this class also increases greatly. The data size also has a great impact on the query time. Therefore, the query time of Q2 and Q3 is larger than that of other query statements. Although Q1 is a single query too, the result set of Q1 is small. So, the query time increase is small. Compared with the data size increased by 10 times, the query time of Q2 and Q3 increased by about 2 times, and the query time was within the acceptable range.

Table 7. Query response time of dataset2 (ms)

Method	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Triple	870	865	603	1581	NAN	306839	NAN	NAN
SW-Store	208	247	387	307	NAN	13893	2822	25493
Jena	114	115	**143**	112	138	121	130	107
RDF2ORDB	**69**	**103**	145	**66**	**78**	**56**	**47**	**98**

Figure 17. The increase ratio of query time



In conclusion, RDF2ORDB achieves good query results when dealing with complex SPARQL statements and highly selective statements.

5.3.2 Analysis of Semantic Retention

Since there is still no standard for whether the storage semantics of RDF(S) are lost, we analyze the semantic retention from two aspects: qualitative analysis and comparative analysis of query results.

(1) Qualitative analysis

According to the formal definition of RDF(S) proposed in Section 2, the semantic information contained in the LUBM dataset is shown in Table 8.

In Table 8, the LUBM dataset contains 43 classes, 7 datatype property and 25 object properties. The LUBM dataset has no datatype defined, so the basic datatype is 0. There are 36 inheritance relationships for classes and 5 inheritance relationship for properties in the LUBM. Then, we analyze the semantic information contained in the object-relational storage structure, which is shown in Table 9.

In addition to the namespace table, property-relational table and property type table, the storage framework proposed in this paper creates a table for each class and property. The property type table contains two tables: *pro_domain* and *pro_range*. So, the database should contain 43 class tables, 32 property tables and four tables defined by the storage framework, for a total of 79 tables which is consistent with the statistical results in Table 9.

The inheritance relationship in Table 9 refers to the number of tables which have parent table. LUBM dataset contains 43 classes. However, *researchAssistant*, *director* and *work* class don't have parent class and there is no property domain includes these classes. Therefore, there should be 40 tables, which is consistent with the statistical results in Table 9.

The storage of properties semantic is relatively simple. The constraint relation and inheritance relation of the properties are stored directly in the property type table and property relation table. The property relation table contains fives records that store each of the five property axioms in Table 6. The *pro_domain* table contains 25 records that store the domain of the properties. *Pro_range* table contains 32 records, which store property types and range values. The data in these two tables are the same as the statistical results in LUBM.

Above all, the object-relational storage framework proposed in this paper can retain RDF Schema semantics completely.

Table 8. Semantic information contained in the LUBM dataset

Name	Number
Class (RC)	43
Datatype property (DataRP)	7
Object property (ObjectRP)	25
Basic datatype (RD)	0
Class axioms (CAxiom)	36
Property axioms (PAxiom)	5

Table 9. The semantic information contained in the object-relational framework

Name	Number
Table (Tab)	79
Inheritance relationship (Inh)	40

Figure 18. Class inheritance relationships in LUBM datasets

Example analysis: By analyzing the RDF Schema file of LUBM, it can be seen that in the definition of LUBM, the subclass relationship of the *Employee* class is shown in Figure 18. Due to the lack of corresponding instances for the *Faculty* class and *Professor* class in the dataset, inferring the *Employee* parent class can obtain all instances of the *Lecturer*, *FullProfessor* and *AssociationProfessor* classes, which is consistent with the experimental results obtained in this paper, indicating that the semantic information in RDF (S) is well preserved. The statement for conducting inference query testing on attributes is as follows:

SELECT ?X ?Y WHERE {?X ub:degreeFrom ?Y}

The semantics of this query statement are to query instances and objects with the property *degreeFrom*. The total number of results obtained from this query statement is 3494. Due to the excessive number of query results, only some of them are displayed, as shown in Figure 19.

(2) Comparative analysis of query results

To test the semantic retention of a storage method, we also need to compare instances in RDF(S) with data stored in the ORDBs. When Q1-Q7 is queried against the database, the number of result sets obtained is exactly the same as that obtained by querying against RDF(S) files. Since the test data set contains tens of thousands to millions of data, it is impossible to compare the query results one by one, so we select Q1 and Q4 query statements which have small result sets for detailed analysis.

The semantics of Q1 query statement is to query all instances that contain the *publicationAuthor* property and the property value of this attribute is http://www.Department0.University0.edu/ AssistantProfessor0 in dataset1. The query results are shown in Figure 20.

In Figure 20, the *x* column represents all instances. The same query criteria are used to query RDF(S) stored in ORDBs. The corresponding SQL statements obtained through conversion in this system are as follows:

SELECT subject FROM publicationauthor WHERE

publicationauthor_value='http://www.Department0.University0.edu/ AssistantProfessor0'

The results obtained by querying the RDF (S) stored in the database using this SQL statement are shown in Figure 21. To show the query results more visually, Figure 19 shows the results of the query in the PostgreSQL visualization tool. By comparing Figure 20 and Figure 21, it can be seen that queries on RDF(S) files are exactly the same as database.

The semantics of the Q4 query statement is to query the property values of the *FullProfessor* instance, its *emailAddress* property, and the *documentDegreeFrom* property The query results need

Figure 19. Query results for instances and objects with the property *degreeFrom*

http://www.Department0.University0.edu/GraduateStudent119	http://www.University687.edu
http://www.Department0.University0.edu/GraduateStudent118	http://www.University321.edu
http://www.Department0.University0.edu/GraduateStudent117	http://www.University952.edu
http://www.Department0.University0.edu/GraduateStudent116	http://www.University98.edu
http://www.Department0.University0.edu/FullProfessor3	http://www.University336.edu
http://www.Department0.University0.edu/FullProfessor3	http://www.University74.edu
http://www.Department0.University0.edu/FullProfessor3	http://www.University112.edu
http://www.Department0.University0.edu/FullProfessor2	http://www.University125.edu
http://www.Department0.University0.edu/FullProfessor2	http://www.University862.edu
http://www.Department0.University0.edu/FullProfessor2	http://www.University653.edu
http://www.Department0.University0.edu/FullProfessor1	http://www.University503.edu
http://www.Department0.University0.edu/FullProfessor1	http://www.University260.edu
http://www.Department0.University0.edu/FullProfessor1	http://www.University882.edu
http://www.Department0.University0.edu/FullProfessor0	http://www.University241.edu
http://www.Department0.University0.edu/FullProfessor0	http://www.University875.edu
http://www.Department0.University0.edu/FullProfessor0	http://www.University84.edu
http://www.Department0.University0.edu/AssociateProfessor9	http://www.University362.edu
http://www.Department0.University0.edu/AssociateProfessor9	http://www.University654.edu
http://www.Department0.University0.edu/AssociateProfessor9	http://www.University494.edu
http://www.Department0.University0.edu/AssociateProfessor8	http://www.University782.edu
http://www.Department0.University0.edu/AssociateProfessor8	http://www.University778.edu
http://www.Department0.University0.edu/AssociateProfessor8	http://www.University715.edu
http://www.Department0.University0.edu/AssociateProfessor7	http://www.University373.edu
http://www.Department0.University0.edu/AssociateProfessor7	http://www.University315.edu
http://www.Department0.University0.edu/AssociateProfessor7	http://www.University872.edu

Figure 20. The results of Q1 query against dataset1

x
<http://www.Department0.University0.edu/AssistantProfessor0/Publication0>
<http://www.Department0.University0.edu/AssistantProfessor0/Publication1>
<http://www.Department0.University0.edu/AssistantProfessor0/Publication2>
<http://www.Department0.University0.edu/AssistantProfessor0/Publication3>
<http://www.Department0.University0.edu/AssistantProfessor0/Publication4>
<http://www.Department0.University0.edu/AssistantProfessor0/Publication5>

Figure 21. The results of Q1 against the database

	subject text
1	http://www.Department0.University0.edu/AssistantProfessor0/Publication4
2	http://www.Department0.University0.edu/AssistantProfessor0/Publication0
3	http://www.Department0.University0.edu/AssistantProfessor0/Publication2
4	http://www.Department0.University0.edu/AssistantProfessor0/Publication3
5	http://www.Department0.University0.edu/AssistantProfessor0/Publication5
6	http://www.Department0.University0.edu/AssistantProfessor0/Publication1

to meet the property values of the *worksFor* property http://www.Department0.University0.edu/FullProfessor. The results obtained by querying dataset1 with Q4 are shown in Figure 22.

In Figure 22, there are three columns, where X represents the *FullProfesosor* instance, Y1 represents the value of the *EmailAddress* property and Y2 represents the value of *doctoralDegreeFrom* property. Using Q4 to query RDF (S) data stored in the database, this system converts the corresponding SQL statements as follows:

```
SELECT
fullprofessor.subject, fullprofessor.emailaddress_value, fullprofessor.doctoraldegreefrom_value
FROM fullprofessor, worksfor
WHERE fullprofessor.subject=worksfor.subject
AND worksfor.worksfor_value='http://www.Department0.University0.edu'
```

The results of this query statement for the database are shown in Figure 23.

The results shown in Figure 22 and Figure 23 are exactly the same. It means that the storage method proposed in this paper can store RDF instances into databases without instances loss.

5.3.3 Discussion and Analysis

RDF (S) storage method performance based on ORDBs: Due to the significant impact of different storage structures on query efficiency, this paper mainly evaluates performance of the proposed method from two aspects: storage performance and query performance. To objectively evaluate the performance of the storage scheme proposed in this paper, three different storage schemes, namely *Triple* (vertical storage), *SW-store* (horizontal storage) and *Jena* (type storage), were used to store RDF(S). The effectiveness of the storage method proposed in this paper was analyzed through comparative experiments. The experimental results are analyzed as follows:

Figure 22. The results of Q4 query against dataset1

X	Y1	Y2
<http://www.Department0.University0.edu/FullProfessor0>	"FullProfessor0@Department0.University0.edu"	<http://www.University241.edu>
<http://www.Department0.University0.edu/FullProfessor1>	"FullProfessor1@Department0.University0.edu"	<http://www.University882.edu>
<http://www.Department0.University0.edu/FullProfessor2>	"FullProfessor2@Department0.University0.edu"	<http://www.University653.edu>
<http://www.Department0.University0.edu/FullProfessor3>	"FullProfessor3@Department0.University0.edu"	<http://www.University112.edu>
<http://www.Department0.University0.edu/FullProfessor4>	"FullProfessor4@Department0.University0.edu"	<http://www.University143.edu>
<http://www.Department0.University0.edu/FullProfessor5>	"FullProfessor5@Department0.University0.edu"	<http://www.University358.edu>
<http://www.Department0.University0.edu/FullProfessor6>	"FullProfessor6@Department0.University0.edu"	<http://www.University135.edu>
<http://www.Department0.University0.edu/FullProfessor7>	"FullProfessor7@Department0.University0.edu"	<http://www.University241.edu>
<http://www.Department0.University0.edu/FullProfessor8>	"FullProfessor8@Department0.University0.edu"	<http://www.University371.edu>
<http://www.Department0.University0.edu/FullProfessor9>	"FullProfessor9@Department0.University0.edu"	<http://www.University730.edu>

Figure 23. The results of Q4 query against database

	subject text	emailaddress_value text	doctoraldegreefrom_value text
1	http://www.Department0.University0.edu/FullProfessor7	FullProfessor7@Department0.University0.edu	http://www.University241.edu
2	http://www.Department0.University0.edu/FullProfessor8	FullProfessor8@Department0.University0.edu	http://www.University371.edu
3	http://www.Department0.University0.edu/FullProfessor9	FullProfessor9@Department0.University0.edu	http://www.University730.edu
4	http://www.Department0.University0.edu/FullProfessor0	FullProfessor0@Department0.University0.edu	http://www.University241.edu
5	http://www.Department0.University0.edu/FullProfessor1	FullProfessor1@Department0.University0.edu	http://www.University882.edu
6	http://www.Department0.University0.edu/FullProfessor2	FullProfessor2@Department0.University0.edu	http://www.University653.edu
7	http://www.Department0.University0.edu/FullProfessor3	FullProfessor3@Department0.University0.edu	http://www.University112.edu
8	http://www.Department0.University0.edu/FullProfessor4	FullProfessor4@Department0.University0.edu	http://www.University143.edu
9	http://www.Department0.University0.edu/FullProfessor5	FullProfessor5@Department0.University0.edu	http://www.University358.edu
10	http://www.Department0.University0.edu/FullProfessor6	FullProfessor6@Department0.University0.edu	http://www.University135.edu

1) The method proposed in this paper takes the longest time to store RDF(S) on the same dataset (as shown in Figure 13), mainly because it requires first parsing the structural information of RDF(S), and then storing RDF(S) in different data tables according to different classes and properties. In future work, consider adopting distributed RDF data storage methods to reduce storage time;
2) On the same dataset, the method proposed in this paper occupies the smallest storage space (as shown in Figure 14), mainly because the method uses a namespace table to store unique prefixes, so that other tables only need to store the prefix ID in the namespace table, avoiding the waste of storage space caused by duplicate prefix IRIs.
3) On the same dataset, the query efficiency of the method proposed in this paper is the highest (as shown in Table 6 and Table 7). The main reason is that when designing the object relational storage model, the class table inherits the property table, so the class table will contain all property fields that define the domain as that class. By querying the class table to obtain the properties of the class, this avoids a large number of null values.

Semantic retention performance of RDF (S) storage method based on ORDBs: Currently, there is no standard for evaluating whether there is semantic loss before and after RDF(S) storage. Therefore, this paper analyzes the semantic retention performance of the proposed RDF(S) storage method through qualitative analysis and comparative analysis of query results before and after RDF(S) storage. The experimental results are analyzed as follows:

1) Qualitative analysis: Based on the proposed RDF(S) formal definition (as shown in Section 3.1), statistical analysis was conducted on the semantic information contained in the LUBM dataset (as shown in Table 8). Meanwhile, analyze the semantic information contained in the created object relational storage structure (as shown in Table 9). The comparison results indicate that the RDF(S) storage method proposed in this paper effectively preserves the semantic information in the RDF Schema;
2) Comparison of query results before and after RDF(S) storage: This paper selects Q1 and Q4 query statements to query the original RDF(S) and RDF(S) stored in ORDBs, respectively. The query results are the same. This further verifying that the RDF(S) storage method proposed in this paper effectively saves semantic information in the RDF Schema.

The experimental results show that the RDF(S) storage method proposed in this paper can effectively preserve semantic information. However, due to the fact that RDF Schema and ORDBs belong to different models, there are certain differences in modeling ideas, implementation methods, and application scenarios between the two, which inevitably leads to some semantic deficiencies.

1) During the process of mapping RDF(S) instances to records in relational database tables, there may be a large number of null values. The main reason for this problem is that the definition of an individual in RDF(S) does not need to explicitly represent all its properties, so it is represented as a null value when mapping to an ORDB.
2) The cardinality of property correspondence cannot be well determined (one-to-one, one-to-many, many-to-one and many-to-many). The constraint of RDF(S) on properties is limited to the *definition domain* and *value domain*, and there is no corresponding metadata to describe the cardinality relationship of properties resulting in semantic loss. This problem can be well solved in OWL language, and the cardinality of property correspondence can be accurately divided through the property function (owl:FunctionalProperty) and inverse function (owl:InverseFunctionalProperty) provided by OWL language.

In summary, viewed from storage time, our proposed storage method is roughly comparable to Triple and SW-store. Jena takes the shortest to store data than other methods and this is a limitation of our method, but the storage time of storage time is still within an acceptable range. More importantly, our method occupies the smallest storage size after storing RDF data and takes almost the shortest in retrieving RDF data. Jointly considering storage time, storage size, query response time and semantic retention, our approach has its advantage, compared with the comparison methods.

6. CONCLUSION

With the wide applications of RDF(S) and the increasing number of RDF(S) data available, effective management of large-scale RDF(S) data are essential. Fully considering the structural advantages of ORDBs, in this paper, we propose an RDF(S) storage paradigm based on the ORDBs. Specifically, we first present the formal definitions of RDF(S) and ORDBs Then, based on the semantic structure contained in RDF(S) and the structure information of the ORDBs, we propose a set of rules that maps RDF(S) data to the ORDBs. We design and implement a prototype system that supports the storage of RDF(S) in the ORDBs. In particular, we use the benchmark LUBM dataset evaluating the performances of storage and query of our storage system. Experimental results show that the RDF(S) storage method proposed in this paper cannot only retain complete semantic information, but also have better query efficiency.

Basically, the ORDB-based RDF(S) storage method proposed in the paper can effectively preserve the semantic information in RDF(S) and meanwhile can improve data query efficiency. In practice, our method is especially suitable for the scenario of managing medium- scale and large-scale RDF data with traditional databases. In this paper, our storage pattern based on the ORDBs is verified only with the benchmark LUBM dataset. However, once the object relational storage model in the database is created, only RDF(S) with the same RDF Schema structure can be stored. For RDF(S) with different schema structures, they cannot be stored in the same database at the moment, and their scalability needs to be improved. Therefore, in future work, we will conduct research from the following aspects:

1) We will study how to uniformly store RDF(S) data with different RDF Scheme structures, and conduct relevant experiments on a large amount of RDF(S) data with different scales and types
2) we plan to utilize the indexing characteristics of ORDBs over the data table structure in the databases. By introducing appropriate indexing and hashing techniques, the query efficiency of RDF(S) storage can be further improved.
3) With the exponential explosive growth of RDF data, the scalability issue of massive RDF(S) store is increasingly important. In our future work, we will improve our approach to support for distributed RDF(S) store and evaluate our approach against some advanced proposals such as Triag (Naacke & Curé, 2020) and WISE (Guo, Gao & Zou, 2020).

REFERENCES

Abadi, D. J., Marcus, A., Madden, S. R., & Hollenbach, K. (2009). SW-Store: A vertically partitioned DBMS for Semantic Web data management. *The VLDB Journal*, *18*(2), 385–406. doi:10.1007/s00778-008-0125-y

Ackere, S. V. (2019). FLIAT, an object-relational GIS tool for flood impact assessment in Flanders, *Belgium. Water (Basel)*, *11*(4), 711–743. doi:10.3390/w11040711

Agrawal, R., Somani, A., & Xu, Y. (2001). Storage and querying of e-commerce data. *Proceedings of the 27th International Conference on Very Large Data Bases*, 149-158.

Alexaki, S. (2001). The ICS-FORTH RDFSuite: managing voluminous RDF description bases. *Proceedings of the 2nd International Workshop on the Semantic Web*.

Arsic, B., Đokić-Petrović, M., Spalević, P., Milentijević, I., Rančić, D., & Živanović, M. (2019). SpecINT: A framework for data integration over cheminformatics and bioinformatics RDF repositories. *Semantic Web*, *10*(4), 795–813. doi:10.3233/SW-180327

Astrova, I., & Kalja, A. (2008). Storing OWL ontologies in SQL3 object-relational databases. *Proceedings of the 8th International Conference on Applied Informatics and Communications*, 99-103.

Atre, M., Srinivasan, J., & Hendler, J. A. (2008). BitMat: a main-memory bit matrix of RDF triples for conjunctive triple pattern queries. *Proceedings of the Poster and Demonstration Session at the 7th International Semantic Web Conference*.

Auzi, A. (2018). Object-relational database structure model and structure optimisation. *Applied Computer Systems*, *23*(1), 28–36. doi:10.2478/acss-2018-0004

Bagui, S. (2003). Achievements and weaknesses of object-oriented databases. *Journal of Object Technology*, *2*(4), 29–41. doi:10.5381/jot.2003.2.4.c2

Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., & Velkov, R. (2011). Owlim: A family of scalable semantic repositories. *Semantic Web*, *2*(1), 1–10. doi:10.3233/SW-2011-0026

Bornea, M. A. (2013). Building an efficient RDF store over a relational database. *Proceedings of the 2013 ACM SIDMOD International Conference on Management of Data*, 121-132. doi:10.1145/2463676.2463718

Broekstra, J., Kampman, A., & van Harmelen, F. (2002). Sesame: A generic architecture for storing and querying RDF and RDF Schema. *Proceedings of the 2002 International Semantic Web Conference*, 54-68. doi:10.1007/3-540-48005-6_7

Chao, C. M. (2007). An object-oriented approach for storing and retrieving RDF/RDFS documents. *Tamkang Journal of Science and Engineering*, *10*(2), 275–286.

Chebotko, A., Lu, S., Jamil, H. M., & Fotouhi, F. (2009). Semantics preserving SPARQL-to-SQL query translation for optional graph patterns. *Data & Knowledge Engineering*, *68*(10), 973–1000. doi:10.1016/j.datak.2009.04.001

Crasso, M., Mateos, C., Zunino, A., & Campo, M. (2012). A programming model for the Semantic Web. *Proceedings of the Second International Conference on Advances in New Technologies, Interactive Interfaces and Communicability*, 208-218. doi:10.1007/978-3-642-34010-9_20

Cudre-Mauroux, P. (2013). NoSQL databases for RDF: an empirical evaluation. *Proceedings of the 12th International Semantic Web Conference*, 310-325. doi:10.1007/978-3-642-41338-4_20

Edwards, R. (2022). *SQL vs. NoSQL: Pros and Cons*. https://dzone.com/articles/sql-vs-nosql-pros-amp-cons-1

Erling, Q., & Mikhailov, I. (2007). RDF support in the virtuoso DBMS. *Proceedings of the 1st Conference on Social Semantic Web*, 59-68.

Fan, T., Yan, L., & Ma, Z. (2020). Storing and querying fuzzy RDF (S) in HBase databases. *International Journal of Intelligent Systems*, *35*(4), 751–780. doi:10.1002/int.22224

Franke, C. (2011). Distributed semantic web data management in HBase and MySQL Cluster. *Proceedings of the 2011 IEEE International Conference on Cloud Computing*, 105-112. doi:10.1109/CLOUD.2011.19

Gregory, V. (2019). Lessons in persisting object data using object-relational mapping. *IEEE Software*, *36*(6), 43–52. doi:10.1109/MS.2018.227105428

Guo, X., Gao, H., & Zou, Z. (2020). WISE: Workload-aware partitioning for RDF systems. *Big Data Research*, *22*, 100161. doi:10.1016/j.bdr.2020.100161

Guo, Y., Pan, Z., & Heflin, J. (2005). LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, *3*(2-3), 158–182. doi:10.1016/j.websem.2005.06.005

Harris, S., & Gibbins, N. (2003). 3store: efficient bulk RDF storage. *Proceedings of the First International Workshop on Practical and Scalable Semantic Systems*, 1-15.

Harris, S., Lamb, N., & Shadbolt, N. (2009). 4store: the design and implementation of a clustered RDF store. *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, 94-109.

Harth, A., Umbrich, J., Hogan, A., & Decker, S. (2007). YARS2: a federated repository for querying graph structured data from the Web. *Proceedings of the 6th International Semantic Web Conference*, 211-224. doi:10.1007/978-3-540-76298-0_16

Khadilkar, V., Kantarcioglu, M., Thuraisingham, B. M., & Castagna, P. (2012). Jena-HBase: a distributed, scalable and efficient RDF triple store. *Proceedings of the 2012 International Semantic Web Conference*.

Khanduja, V., & Chkraverty, S. (2019). A generic watermarking model for object relational databases. *Multimedia Tools and Applications*, *78*(19), 28111–28135. doi:10.1007/s11042-019-07932-3

Levandoski, J. J., & Mokbel, M. F. (2009). RDF data-centric storage. *Proceedings of the 2009 IEEE International Conference on Web Services*, 911-918. doi:10.1109/ICWS.2009.49

Ma, R., Jia, X., Cheng, J., & Angryk, R. A. (2016). SPARQL queries on RDF with fuzzy constraints and preferences. *Journal of Intelligent & Fuzzy Systems*, *30*(1), 183–195. doi:10.3233/IFS-151745

Ma, Z., Capretz, M. A. M., & Yan, L. (2016). Storing massive Resource Description Framework (RDF) data: A survey. *The Knowledge Engineering Review*, *31*(4), 391–413. doi:10.1017/S0269888916000217

Ma, Z., Lin, X., Yan, L., & Zhao, Z. (2018). RDF keyword search by query computation. *Journal of Database Management*, *29*(4), 1–27. doi:10.4018/JDM.2018100101

Michel, F., Faron-Zucker, C. & Montagnat, J. (2019). Bridging the Semantic Web and NoSQL worlds: generic SPARQL query translation and application to MongoDB. *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, *40*, 125-165.

Naacke, H., & Curé, O. (2020). Triag, a framework based on triangles of RDF triples. *Proceedings of The International Workshop on Semantic Big Data*, 1-6. doi:10.1145/3391274.3393634

Neumann, T., & Weikum, G. (2010). The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, *19*(1), 91–113. doi:10.1007/s00778-009-0165-y

Pan, Z., & Heflin, J. (2003). DLDB: extending relational databases to support semantic Web queries. *Proceedings of the First International Workshop on Practical and Scalable Semantic Systems,* 109-113.

Rodriguez-Muro, M., & Rezk, M. (2015). Efficient SPARQL-to-SQL with R2RML mappings. *Journal of Web Semantics*, *33*, 141–169. doi:10.1016/j.websem.2015.03.001

Rubio-Largo, A., Vanneschi, L., Castelli, M., & Vega-Rodríguez, M. A. (2017). Using biological knowledge for multiple sequence aligner decision making. *Information Sciences*, *420*, 278–298. doi:10.1016/j.ins.2017.08.069

Shen, K., Hu, S., & Tzeng, G. (2017). Financial modeling and improvement planning for the life insurance industry by using a rough knowledge based hybrid MCDM model. *Information Sciences*, *375*, 296–313. doi:10.1016/j.ins.2016.09.055

Smiatacz, M. (2018). Normalization of face illumination using basic knowledge and information extracted from a single image. *Information Sciences*, *469*, 14–29. doi:10.1016/j.ins.2018.08.034

Song, D., Schilder, F., Hertz, S., Saltini, G., Smiley, C., Nivarthi, P., Hazai, O., Landau, D., Zaharkin, M., Zielund, T., Molina-Salgado, H., Brew, C., & Bennett, D. (2019). Building and querying an enterprise knowledge graph. *IEEE Transactions on Services Computing*, *12*(3), 356–369. doi:10.1109/TSC.2017.2711600

Stefani, E., & Hoxha, K. (2018). Implementing triple-stores using NoSQL databases. *Proceedings of the 3rd International Conference on Recent Trends and Applications in Computer Science and Information Technology*, 86-92.

Wilkinson, K., Sayers, C., Kuno, H. A., & Reynolds, D. (2003). Efficient RDF storage and retrieval in Jena2. *Proceedings of the 2003 Semantic Web and Databases Workshop*, 131-150.

Wu, G., Li, J. Z., Hu, J. Q., & Wang, K. H. (2009). System π: A native RDF repository based on the hypergraph representation for RDF data model. *Journal of Computer Science and Technology*, *24*(4), 652–664. doi:10.1007/s11390-009-9265-9

Xiong, C., Power, R., & Callan, J. (2017). Explicit semantic ranking for academic search via knowledge graph embedding. *Proceedings of the 26th International Conference on World Wide Web*, 1271-1279. doi:10.1145/3038912.3052558

Zhang, F., Ma, Z. M., & Li, W. J. (2015). Storing OWL ontologies in object-oriented databases. *Knowledge-Based Systems*, *76*(1), 240–255. doi:10.1016/j.knosys.2014.12.020

Zheng, W., Cheng, H., Yu, J. X., Zou, L., & Zhao, K. (2019). Interactive natural language question answering over knowledge graphs. *Information Sciences*, *481*, 141–159. doi:10.1016/j.ins.2018.12.032

Zhou, G. (2017). Graph-based knowledge reuse for supporting knowledge-driven decision-making in new product development. *International Journal of Production Research*, *55*(23), 7187–7203. doi:10.1080/00207543.2017.1351643

ENDNOTES

1 http://sparqlcity.com/
2 https://www.marklogic.com/
3 http://clarkparsia.com/
4 http://www.cs.ox.ac.uk/isg/tools/RDFox/
5 https://docs.sesam.io/rdf-support.html
6 http://dslam.cs.umd.edu/swstore/index.html
7 https://jena.apache.org/
8 https://www.w3.org/TR/json-ld/
9 https://swat.cse.lehigh.edu/projects/lubm/
10 https://www.w3.org/TR/sparql11-query/

Ruizhe Ma is currently an assistant professor in the Richard A. Miner School of Computer & Information Sciences at the University of Massachusetts Lowell, USA. She received her PhD degree from the Department of Computer Science at Georgia State University. Dr. Ma's research focuses on data mining, machine learning, big data analytics, graph mining, and K-12 education.