OntoDS: An Ontology-Aware Distributed Storage Scheme for RDF Graphs

Baozhu Liu[1], Xin Wang[1,2](✉), Yajun Yang[1,2], and Yunpeng Chai[3]

[1] College of Intelligence and Computing, Tianjin University, Tianjin, China
{liubaozhu,wangx,yjyang}@tju.edu.cn
[2] Tianjin Key Laboratory of Cognitive Computing and Application, Tianjin, China
[3] School of Information, Renmin University of China, Beijing, China
ypchai@ruc.edu.cn

Abstract. With the development of the Semantic Web, the amount of RDF data has been increasing rapidly. It is no longer feasible to store entire data sets on a single machine, and still be able to access the data at reasonable performance. Consequently, the requirement for clustered RDF database systems is becoming more and more important. At the same time, the native storage scheme of RDF data is less mature in many aspects compared with relational storage scheme. SQL-on-Hadoop is a distributed relational database engine for big data with many factors, which uses Hadoop to improve the fault tolerance of the system and is fully transactional. However, currently, there is no SQL-on-Hadoop relational database that realizes a subsystem for RDF data storage. In this paper, we propose an Ontology-aware Distributed Storege scheme for RDF, called OntoDS, which modifies the relational RDF data storage scheme DB2RDF to build a novel scheme for RDF data and optimizes the partitioning of RDF graphs by distributing RDF triples based on ontologies to meet the need for RDF graph data storage and query load. The experimental results on the benchmark datasets show that our distributed RDF storage scheme is about 1–1.5 times faster than the state-of-the-art native storage schemes.

Keywords: RDF data storage · RDF graph · DB2RDF

1 Introduction

Among the data models of knowledge graphs, the Resource Description Framework (RDF [1]) is a model for representing Web resources, which has become a standard format for knowledge graphs and is widely used. With the development of the Semantic Web, RDF format is gaining widespread acceptance and the amount of RDF data has been dramatically increasing. The number of triples of the latest 2016-10 version of the DBpedia [2] dataset has reached 13 billion. With the rapid rise of the data volume of RDF graphs, it is no longer feasible to store entire data sets on a single machine. In order to solve the scalability problem of the RDF storage scheme on a single machine, distributed RDF storage scheme has become an inevitable option. On the other hand, native storage schemes of RDF data are less mature in many aspects compared with the corresponding relational versions. Thus, we choose relational storage schemes rather than the native ones.

Although many models have been proposed to store RDF graphs [3] (e.g., triple table, horizontal table, property table, vertical partitioning [4], sextuple indexing, DB2RDF [5], and SQLGraph [6]), the existing solutions are implemented on a single machine, not in a distributed environment. SQL-on-Hadoop is a kind of data management technology based on Hadoop [7], which is a data query and storage mechanism using SQL as its query language. SQL-on-Hadoop architecture is suitable for the storage of large-scale RDF graph data due to its high degree of parallelism, robustness, reliability, and scalability while running on heterogeneous commodity hardware. Therefore, a distributed database with SQL-on-Hadoop architecture can be used to solve the storage problem of RDF graph data.

Based on SQL-on-Hadoop, some distributed RDF storage systems are proposed. The system details will be introduced in Sect. 5. Among these systems, none of them combines MPP features with ontology-aware distribution of RDF graphs, which can significantly accelerate queries.

To speed up the queries over RDF graphs in a distributed environment, it is obvious that a reasonable RDF graph distribution method needs to be first considered. Since many RDF queries depend on ontology information, it is beneficial to realize an ontology-aware data distribution for RDF graphs in a distributed cluster. Unlike the random distribution of RDF triples in the existing systems, our OntoDS storage scheme takes full advantage of ontologies associated with RDF graphs to partition and store RDF triples in a semantic-aware manner.

In this paper, we focus the distributed storage scheme of RDF graphs and propose OntoDS, which is an ontology-aware distributed RDF storage scheme. Meanwhile, based on the SQL-on-Hadoop infrastructure, we have developed a prototype system that implements the OntoDS storage scheme and supports efficient RDF query processing on top of OntoDS.

Our contributions can be summarized as follows:

(1) We propose a novel relational storage scheme for RDF data with five relations, which is flexible to handle dynamic RDF schemas, as it does not require schema changes when RDF triples being inserted.
(2) The prefix encoding used to record ontology information not only facilitates the distribution of RDF data, but also keeps the hierarchical information of ontologies. Compared with type-oriented methods, which can only provide the nearest ontology of entities, the prefix encoding can give more helpful information during queries.
(3) Extensive experiments were conducted to verify the scalability and efficiency of OntoDS. The experimental results show that OntoDS is about 1–1.5 times faster than the state-of-the-art native storage schemes.

The rest of this paper is organized as follows. Section 2 provides an overview of OntoDS. In Sect. 3, the distribution method over RDF graph of OntoDS is introduced. Section 4 shows experimental results on benchmark datasets. Section 5 briefly reviews related work. Finally, we conclude in Sect. 6.

2 RDF over Relational

There have been many attempts to shred RDF data into relational models. DB2RDF [5] is one of the entity-oriented alternatives, however it does not distribute data in a semantic-aware way, which can reduce data shuffle and further accelerate queries. Thus, we modify DB2RDF by adding ontology information for data distribution to provide a suitable scheme for distributed environment.

2.1 The OntoDS Storage Scheme

OntoDS is composed of five relations, including Direct Primary Hash (`DPH`), Reverse Primary Hash (`RPH`), Direct Secondary Hash (`DS`), Reverse Secondary Hash (`RS`), and `TYPES`, as is depicted in Fig. 1. The `DPH` and `DS` relations essentially encode the outgoing edges of an entity, in other words, the entities in `DPH` represent subjects in RDF triples. Meanwhile, the `RPH` and `RS` relations encode the incoming edges of an entity, which means the entities in `RPH` represent objects in RDF triples. In order to distinguish the columns of `DPH` and `RPH`, we use different subscripts on these columns, e.g., $value_{m1}$ and $value_{n1}$. `RPH` and `RS` relations are added to facilitate object-given queries. The `TYPES` relation records the codings of all ontologies.

In our scheme, `DPH` is a wide relation, in which each tuple stores a subject s in the entry column, with its ontology information stored in the *type* colomn and all its associated predicates and objects stored in the $pred_i$ and val_i columns, respectively $(0 \leq i \leq k)$. If subject s has more than k predicates, the extra predicates are spilled to another tuple and process continues until all the predicates for s are stored. When it comes to multi-valued predicates, a new unique identifier is assigned as the value of the predicate in `DPH` relation. Then, the identifier is stored in the `DS` relation along with its real predicate values. `RPH` and `RS` works in the same way as `DPH` and `DS`. The example of the scheme is shown in Fig. 3. The related RDF graph is shown in Fig. 2(a).

OntoDS treats the columns of a relation as flexible storage locations that are not pre-assigned to any predicate, but predicates are assigned to them dynamically, during insertion. The assignment ensures that a predicate is always assigned to the same column or more generally the same set of columns.

We refer to a query with common subject or object and its adjacent nodes as a *star query*. To execute query over OntoDS, after the triples with the same subject or object being merged, the query rewriter will construct a single SQL SELECT-statement for each star query. The ontology of entities can be generated by joining `DPH` (resp. `RPH`) with `TYPES`. If the subjects (resp. objects) are given in the queries, we use `DPH` (resp. `RPH`) to get the result. When star query involving

DPH

type	entry	spill	pred$_{n1}$	val$_{n1}$	pred$_{n2}$	val$_{n2}$	...	pred$_{nn}$	val$_{nn}$

RPH

type	entry	spill	pred$_{m1}$	val$_{m1}$	pred$_{m2}$	val$_{m2}$	...	pred$_{mm}$	val$_{mm}$

DS

l_id	elm

RS

l_id	elm

TYPES

code	type

Fig. 1. OntoDS storage scheme.

multi-valued predicates, the SQL statement will join `DPH` (resp. `RPH`) with `DS` (resp. `RS`) together to product the actual objects (resp. subjects) of a subject (resp. object) entity.

The number of columns in `DPH` and `RPH` relations is decided by predicate inteference graph coloring, and predicates along with their corresponding objects are inserted by string hash functions. The details will be explained in next subsection.

2.2 Data Insertion

The objective of the OntoDS scheme is to dynamically assign predicates of a given dataset to columns such that:

(1) the total number of columns used across all subjects is minimized;
(2) for a subject, the probability to mapping two different predicates into the same column is minimized to reduce *spill*.

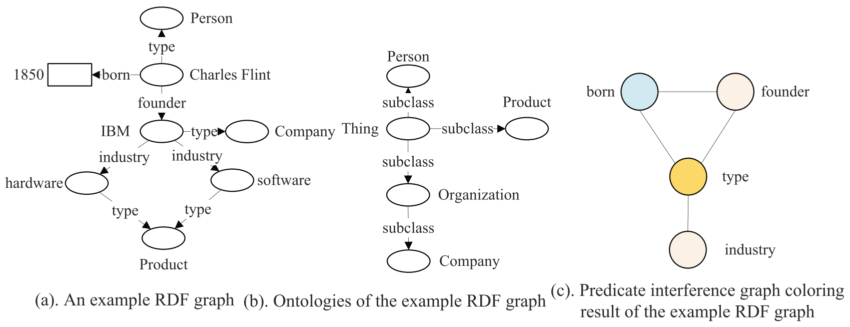
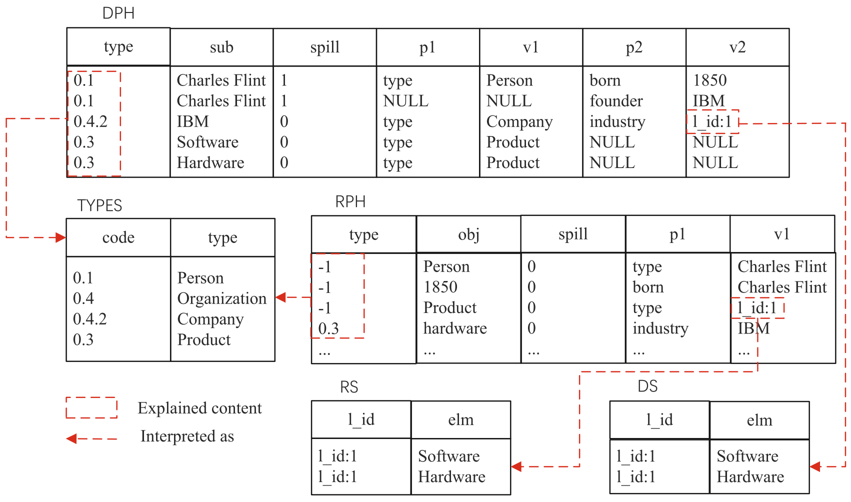


Fig. 2. An example typed RDF graph.



DPH

type	sub	spill	p1	v1	p2	v2
0.1	Charles Flint	1	type	Person	born	1850
0.1	Charles Flint	1	NULL	NULL	founder	IBM
0.4.2	IBM	0	type	Company	industry	l_id:1
0.3	Software	0	type	Product	NULL	NULL
0.3	Hardware	0	type	Product	NULL	NULL

TYPES

code	type
0.1	Person
0.4	Organization
0.4.2	Company
0.3	Product

RPH

type	obj	spill	p1	v1
-1	Person	0	type	Charles Flint
-1	1850	0	born	Charles Flint
-1	Product	0	type	l_id:1
0.3	hardware	0	industry	IBM
...	...	...	...	...

RS

l_id	elm
l_id:1	Software
l_id:1	Hardware

DS

l_id	elm
l_id:1	Software
l_id:1	Hardware

Fig. 3. Example scheme for typed RDF graph.

The same column cannot store different predicates of the same subject. Formal definition is given as follows:

Definition 1 (Predicate Mapping). *A Predicate Mapping is a function: URL $\rightarrow \mathcal{N}$, and the domain of which is URIs of predicates and the range of which is natural numbers between 0 and maximum m, m is the largest allowed number on a single database row.*

$\mathcal{N}$ can be determined by predicate interference graph coloring, the definition of predicate inteference graph can be formally given as:

Definition 2 (Predicate Inteference Graph). *G_D is a Predicate inteference graph for a specific dataset D such that:*

$$V_D = \{p \mid \langle s, p, o\rangle \in D\} \tag{1}$$

$$E_D = \{\langle p_i, p_j\rangle \mid \langle s, p_i, o\rangle \in D \land \langle s, p_j, o\rangle \in D\} \tag{2}$$

Correspondingly, the predicate inteference graph coloring problem can be defined as Definition 3. In a predicate interference graph, where predicates with the same subject are connected, the nodes connected cannot be assigned to the same color. The coloring result of the example RDF graph's predicate interference graph Fig. 2(a) is depicted in Fig. 2(c)

Definition 3 (Predicate Inteference Graph Coloring). *For specific predicate inteference graph $G = \langle V, E\rangle$, its predicate inteference graph coloring result C is a maping from vertex v to color c, that:*

$$M(G, C) = \{\langle v, c\rangle \mid v \in V \land c \in C \land (\langle v_i, c_i\rangle \in M \land \langle v, v_i\rangle \in E \rightarrow c \neq c_i)\} \tag{3}$$

Since graph coloring is an NP problem [8], we choose the state-of-the-art heuristic algorithm *Welsh-Powell* [9] graph coloring algorithm, whose basic idea is shown in Algorithm 1. The details of this algorithm are as follows:

(1) All vertices in the graph G are sorted in descending order of their degrees.
(2) We assign the first color to the first vertex, and then color the others according to the order. In the same iteration of coloring, colored vertex is not adjacent to each other.
(3) The remaining ordered vertices that is not colored is traversed until all the vertices are colored.

Algorithm 1: Interference graph coloring

```
Data: predicate interference graph $G' = \langle V', E' \rangle$
Result: graph coloring result $color\_count$
1  $color\_count := 0$;                          // the counts for used colors
2  $C := \emptyset$;                             // the set for colored vertices
3  for $each\ v_i \in V'$ do
4      if color($v_i$) = $false$ then
           // this vertex is not colored
5          color($v_i$) := true;
           // color this vertex
6          $color\_count := color\_count + 1$;
           // the counts for used colors add one
7          $C := C \cup \{v_i\}$;
           // include it into the set for colored vertices
8          for $v_j \in V'$ do
9              if not_neightbor_of($C$) then
                   // $v_j$ is not connected to $v_i$
10                 $C := C \cup \{v_j\}$;
                   // include it into the set for colored vertices
11                 color($v_j$) := true ;
                   // color the vertex
12             return $color\_count$
```

The result of the predicate interference graph coloring guides us to build the DPH and RPH relations, and the insertion of the relations is determined by string hash functions. To minimize column collisions, eight string hash functions were selected, and the calculation method of selected ones are irrelevant. The specific workflow is shown in Algorithm 2.

RDF data insertion using eight string hash functions can be considered as the process of predicate combination composition, which is formally defined as follows:

Definition 4 (Predicate Mapping Composition). *A Predicate Mapping Composition, defines a new predicate mapping that combines the column numbers from multiple predicate mapping functions* $f_1, ...f_n$*:*

$$f_{m,1} \oplus f_{m,2} \oplus ... \oplus f_{m,n} \equiv \{v_1, ..., v_n \mid f_{m,i}(p) = v_i\} \tag{4}$$

For each hash function, the random strings composed of letters and numbers are calculated. The effect of `BKDRHash` is the best. `APHash` is not as good as `BKDRHash`, moreover, is also worse than `DJBHash`, `JSHash`, `RSHash`, and `SDBMHash`. `PJWHash` and `ELFHash` are the worst. Except for `PJWHash` and `ELFHash`, the number of hash collisions per 10,000 strings is about 2 to 3 for each hash function, and `PJWHash` and `ELFHash` are about 30 per 10,000. It can be observed that the effects of selected eight hash functions are desirable.

Algorithm 2: String hash insertion

```
Data: subject $s$ with its predicates $P(s)$ and predicates' corresponding
      objects $O(s)$
Result: $color\_count$ pairs of predicate-object array $PO$ after hash insertion
1 for each $pred_i \in P(s)$ do
      // string hash results for $pred_i$
2     $v_i :=$ HASH($pred_i$) % $color\_count$;
      // 'HASH' refers to the eight selected string hash
         functions, i.e., SDBMHash, RSHash, JSHash, PJWHash,
         ELFash, BKDRHash, DJBHash, and APHash
3     if $PO[v_i] = NULL$ then $PO[v_i] := \{pred_i, obj_i\}$;
4     else split_to_another_row();
      // hash collision occurs in all functions, and the data
         have to be inserted into another row in DPH
```

The existing systems randomly distribute data by entities, so that all data with the same entity will be distributed to the same node. This distribution approach is easy to implement, but does not consider the real-world query needs. In order to accelerate type-related queries, which is common in real-world queries, OntoDS takes full advantage of ontologies associated with RDF graphs to partition and store RDF triples in a semantic-aware manner. In the next Section, we will explain the RDF graph distribution method of OntoDS in detail.

3 Ontology-Aware RDF Graph Distribution

OntoDS, which is shown in Fig. 1, records the ontology information of each entity in the corresponding type column of `DPH` or `RPH` relation, and creates a `TYPES` relation to store each type and its encoding. `DPH` and `RPH` are distributed by type column. Therefore, entities of the same type are assigned to the same node. As we all know, in a distributed environment, we should reduce communication between nodes as much as possible, since communication is the most time consuming process. By ontology-aware distribution, type-related queries will be greatly accelerated, since queries are processed locally, and data shuffle is significantly reduced.

Unlike type-oriented methods, OntoDS records the entity's ontology information for data distribution and querying rather than creates a separate relation for each type. This approach avoids the disadvantages of data sparseness in the type-oriented method, however still easy to obtain the ontology information, when it is needed in queries. For the query workloads provided by many benchmark datasets always first give the type of the involved entity, recording the ontology information of the entity, we can immediately limit the range of data to some nodes in the distributed environment.

3.1 RDF Ontology Information

The IRI (Internationalized Resource Identifiers) of an RDF resource contains a namespace prefix indicating the classes or attributes of the RDF entity, and RDF vocabulary indicates the meaning of these classes. Common RDF vocabularies included FOFA, Dublin Core, Schema.org etc. RDF resources are divided into various classes. Each class has its own instance, and the collection of instances is an extension of a class. Two classes may have the same set of instances but be different classes, and a class can also be its own extension. A class can have its subclasses, so RDF is actually a hierarchical structure. The example RDF ontology hierarchical structure in Fig. 4 is extracted from Lehigh University Benchmark (LUBM) [10]. The toppest ancestor of each ontology is *owl:Thing*.

As is shown in Fig. 4, *Professor* is constituted by *Dean*, *Chair*, *AssistantProfessor*, *FullProfessor*, *AssociateProfessor*, and *VisitingProfessor*. A RDF dataset could have large number of types (e.g. the DBpedia ontology contains 150K types), but not all types appear in the actual data. For example, there are 41 types in the OWL file (the file format to store ontology information of a RDF graph) of LUBM datasets, but only 13 of them are actually used. Thus, we need to record type information based on actual data, not on priori knowledge to minimize records.

The ontology semantic distribution method is easy to implement, as DB2RDF provides us with the convenience of clustered data based on entities. We only need to capture the ontology information of every entity. When extracting various predicates along with their values of an entity, we do not need to concern about the entity's ontology information, and vice versa. So, the whole process of OntoDS can be divided into two separate processes, easy to operate. The type information of the entity is determined by the triple with the predicate *type*, and the hierarchical ontology information needs to be found in the OWL file, i.e., RDFS (Resource Description Framework Scheme) and RDF often stored in different files, which in turn facilitates our separate storage process.

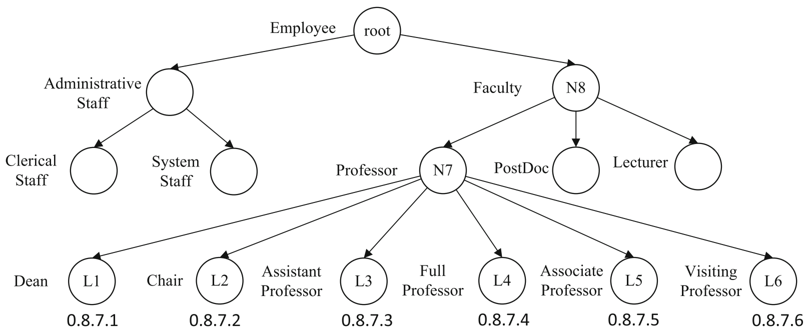


Fig. 4. RDF hierarchical structure and prefix encoding.

3.2 Type Hierarchy Coding

The type hierarchical information of RDF graph Fig. 2(*a*) is shown in Fig. 2(*b*). All entities with no type information or strings are coded like '-1', while the highest level type *Thing* is encoded as '0'. Every hierarchy of ontologies except *Thing* will be recorded in `TYPES` relation. Figure 4 shows an example of type hierarchy coding. Each time, we only focus on the ontology encoding of one leaf node, recursively upward encode the nodes until the highest level ontology is met. The ontology encode method is shown in Algorithms 3 and 4.

Algorithm 3: Ontology encoding

```
  Data: the types encountered: $T$
  Result: the codes of types: $type.code$
1 for $type_i \in T$ do
      // get the code for every type in the set
2     return $type_i.code$ := getcode($type_i$)
3 return $type.code$
```

3.3 Queries on Ontologies

With the ontologies of entities recorded, we should change the query statements to take full advantages of the storage scheme. When it comes to a specific query, we can first point out the type of the involved entities to restrict the entities to some node rather than the whole cluster, which will reduce data shuffle and accelerate queries.

Algorithm 4: getcode($type_i$)

```
Data: the type need to be encoded: $type_i$, the types encountered: $T$
Result: the code for the $type_i$: $type_i.code$
1 if $type_i$ is_subclass_of $type_j$ then
2     $type_j.key := T.size$;
      // all nodes are numbered in order of appearance
3     $T := T \cup \{type_j\}$;
      // include this type into the set $T$
4     return getcode($type_j$) + $type_i.key$;
      // '+' refers to string concatenation
5 else return $type_i.key$;
```

The most typical type-related query is just like: *query the number of publications of A*. We can alter the query like: *query the number of entities whose type is Publication, and whose author is A*.

The best query order for OntoDS should be: (1) find the ontology code of the queried entity in the TYPES relation, (2) query the entity according to the ontology in the corresponding DPH or RPH relation, and (3) find the required data according to the filter information. This query order maximizes the query efficiency of type-related aggregate queries.

4 Experiments

In this section, a thorough experimental study on the RDF data benchmark dataset is conducted to evaluate the performance of OntoDS, using HAWQ [11] as our relational backend. The tested systems are deployed on a 4-node cluster, of which 3 nodes are used for segments and DataNodes, and 1 node is used for master and NameNode. Each node has 4-core, Intel(R) Core(TM) i7-6700 CPU @ 3.40 GHz system, with 16 GB of memory, running 64-bit Linux, and 50 GB hard disk. We conducted experiments on LUBM [10]. Each query was issued 4 times, the first run of which was discarded, and 3 consecutive runs after the first run were used for the average result.

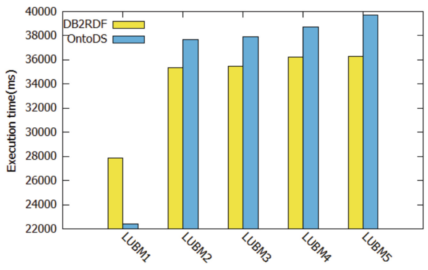
4.1 Datasets

LUBM consists of a university domain ontology, along with customizable and repeatable synthetic data. As the basic idea of OntoDS is to maximize the efficiency of type-related queries, we choose some other query statements instead of using the benchmark queries LUBM provides. The chosen queries are listed in Appendix A (*lubmc* refers to the schema of RDF graph using DB2RDF, *lubmt* refers to that using OntoDS).

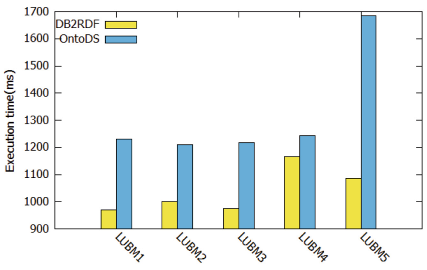
4.2 Experimental Results

Main Results. In general, the experimental results show that OntoDS is both efficient and scalable. Data insertion and deletion can be completed in a short time. The results show that OntoDS is suitable to store RDF data in a distributed environment. The prototype system has certain practical significance.

Data Insertion and Deletion. Although OntoDS needs more time than DB2RDF in data insertion and deletion, their time costs are on the same order of magnitude, thus are comparative, as is shown in Fig. 5. OntoDS can achieve promising insertion efficiency on small RDF data sets. As the amount of RDF data grows, the type information is more dispersed, and the insertion time on OntoDS grows faster than DB2RDF. Although OntoDS is not dominant in the data insertion and deletion, the slight overhead paid on the insertion is worthwhile compared to the gain in the efficiency of queries.

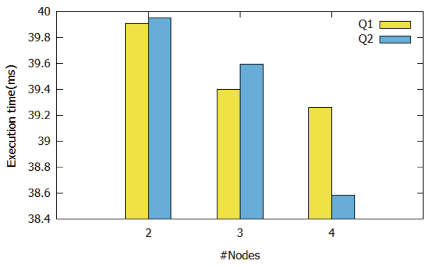


(a) Data insertion

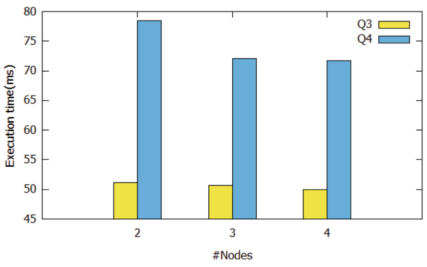


(b) Data deletion

Fig. 5. The experimental results of data insertion and deletion on LUBM datasets.



(a) The results of Q1 and Q2



(b) The results of Q3 and Q4

Fig. 6. The experimental results of scalability.

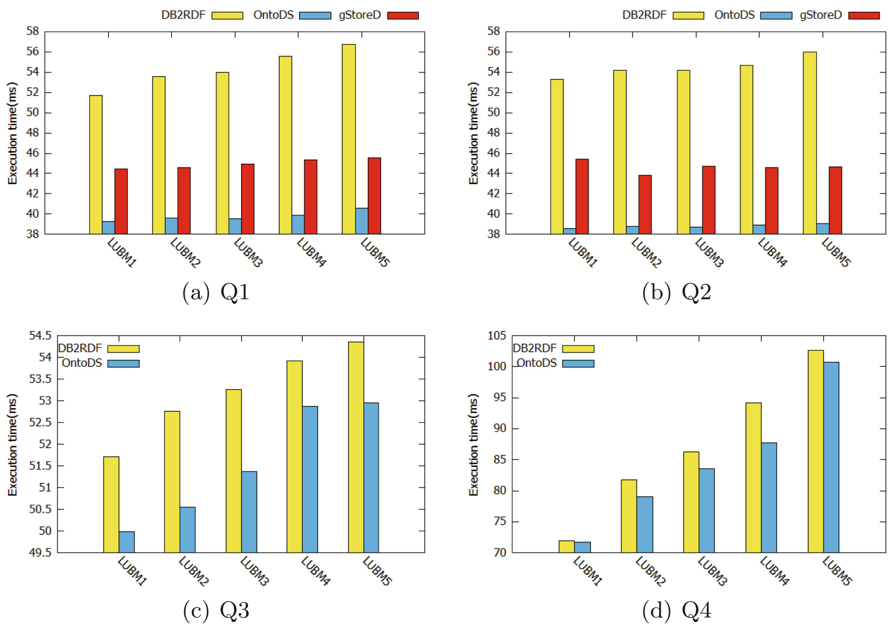


Fig. 7. The experimental results of efficiency on LUBM datasets.

Query Speed. We have selected four kinds of type-related queries: (1) directly about type information, (2) related to the type information with some filtering conditions, and (3) two kinds of queries that do not directly related to the type information but the queried data is clustered by type. Each query is executed one by one to minimize the impact on query time from external factors. Due to the dynamic changes of the network, the query time may not always follow a proportional relationship, while the overall trend remains. There are significant differences between DB2RDF and OntoDS on queries that directly related to type information, such as Q1 and Q2. In the queries that are not directly related to the types, we can see gaps between the two schemes, and OntoDS is faster. In these queries, OntoDS is on average 1 times faster than DB2RDF, and the best one (Q2) can be almost 1.5 times faster than DB2RDF.

We also compared OntoDS with the state-of-the-art distributed RDF storage system gStoreD [12]. Since gStoreD uses SPARQL rather than SQL as its query language, it does not directly support the execution of Q3 and Q4. The query results of Q1 and Q2 are shown in Fig. 7(a) and (b). We can conclude that OntoDS is faster than gStoreD on type-related queries.

Scalability. To validate the scalability of OntoDS, we conducted experiments on LUBM, varying the number of nodes from 2 to 4. The results depicted in Fig. 6 suggests that for a fixed dataset, the execution time is near-linearly decreased as the cluster size increases, in other words, OntoDS is scalable and flexible.

5 Related Work

Single Node RDF Data Storage. In the field of single node RDF data storage, there have been many attempts to shred RDF data into the relational model. The most straightforward solution is to use the characteristics of the RDF triples to store in a Triple Table. This solution takes up too much storage space. Even if it only stores small amount of RDF data, the table needs to have many rows. Another approach, Horizontal Table records all predicates and objects of a subject in one tuple. This solution does not save much space, because considering the varieties of predicates, this horizontal table can have numerous columns while each subject has fewer predicates, so the table have a lot of empty items. Except the schemes above, several storage schemes focus on the type characteristics of the RDF graph data, e.g., Property Table creates tables based on the types of the subjects; Vertical Partitioning creates tables based on the types of predicates [4]. Type-oriented approaches perform simple classifications to reduce the number of rows and empty items in the table. However, they require schema changes as new RDF types are encountered, which is unbearable. Sextuple Indexing storage scheme is created in order to facilitate various join operations, which establishes six tables by all six forms of triplets. This scheme sacrifices storage space while optimizing for queries. DB2RDF [5] is an entity-oriented alternative, which avoids both the skinny relation of the triple table, and the schema changes required by type-oriented approaches. Nevertheless, as mentioned above, DB2RDF is not suitable for a distributed environment.

Distributed RDF Data Storage. In the field of distributed RDF data storage, based on SQL-on-Hadoop, some distributed RDF storage systems are proposed. The H2RDF+ [13] system realizes Sextuple Indexing based on the HBase distributed repository. This approach trades much storage space for get a better query effect, while saving storage space as much as possible is the original intention of our scheme. Sempala [14] is an RDF graph data query engine based on the distributed SQL-on-Hadoop database Impala and Parquet columnar file format. Nevertheless, Sempala is not a relational storage scheme. Stylus [15] is a distributed RDF graph repository that uses strong type information to build optimized storage schemes and query processing. The underlying layer is based on a key-value library. gStoreD [12] is an RDF graph storage scheme that can optimize graph partitioning and store RDF graph based on query load. However, there is no consideration of ontology information in Stylus and gStoreD.

To the best of our knowledge, OntoDS is the first distributed RDF storage scheme to consider ontology information and distributed situations.

6 Conclusion

This paper presented OntoDS, an ontology-aware distributed storage scheme for RDF graphs, and implemented a prototype system of OntoDS based on HAWQ. OntoDS has additional benefits for type-related queries in distributed environment, as it reduces data shuffle between nodes. The experimental results on the benchmark datasets show that our distributed RDF storage scheme is both efficient and scalable, which is 1–1.5 time faster than the state-of-the-art schemes.

Acknowledgments. This work is supported by the National Natural Science Foundation of China (61572353, 61402323) and the Natural Science Foundation of Tianjin (17JCYBJC15400).

A Appendix

A.1 Queries for DB2RDF

```
Q1: SELECT COUNT(*) FROM lubmc.dph WHERE lubmc.dph.p1='type'
    AND lubmc.dph.v1='AssistantProfessor';
Q2: SELECT COUNT(*) FROM lubmc.dph WHERE lubmc.dph.p1='type'
    AND lubmc.dph.v1='AssistantProfessor'
    AND lubmc.dph.p2='mastersDegreeFrom'
    AND lubmc.dph.v2 ='http://www.University389.edu';
Q3: SELECT COUNT(*) FROM lubmc.rph
    WHERE lubmc.rph.obj LIKE '%Lecturer%' ;
Q4: SELECT COUNT(*) FROM lubmc.dph JOIN lubmc.rph
    ON lubmc.dph.sub=lubmc.rph.obj;
```

A.2 Queries for OntoDS

```
Q1: SELECT COUNT(*) FROM lubmt.dph
    WHERE lubmt.dph.type='0.17.16.15.2';
Q2: SELECT COUNT(*) FROM lubmt.dph
    WHERE lubmt.dph.type='0.17.16.15.2'
    AND lubmt.dph.p2='mastersDegreeFrom'
    AND lubmt.dph.v2 ='http://www.University389.edu';
Q3: SELECT COUNT(*) FROM lubmt.rph
    WHERE lubmt.rph.obj LIKE '%Lecturer%' ;
Q4: SELECT COUNT(*) FROM lubmt.dph JOIN lubmt.rph
    ON lubmt.dph.sub=lubmt.rph.obj;
```

A.3 Queries for gStoreD

```
Q1: SELECT ?a WHERE {?a type AssistantProfessor.};
Q2: SELECT ?a WHERE {?a type AssistantProfessor. ?a mastersDegreeFrom
    http://www.University389.edu.};
```

References

1. W3C: RDF 1.1 concepts and abstract syntax (2014)
2. Lehmann, J., et al.: DBpedia-a large-scale, multilingual knowledge base extracted from Wikipedia. Semant. Web **6**(2), 167–195 (2015)

3. Wang, X., Zou, L., Wang, C., Peng, P., Feng, Z.: Research on knowledge graph data management: a survey. Ruan Jian Xue Bao/J. Softw. **30**(7), 2139–2174 (2019). (in Chinese). http://www.jos.org.cn/1000-9825/5841.htm
4. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: SW-Store: a vertically partitioned DBMS for Semantic Web data management. VLDB J. **18**(2), 385–406 (2009)
5. Bornea, M.A., et al.: Building an efficient RDF store over a relational database. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 121–132. ACM (2013)
6. Sun, W., Fokoue, A., Srinivas, K., Kementsietsidis, A., Hu, G., Xie, G.: SQLgraph: an efficient relational-based property graph store. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1887–1901. ACM (2015)
7. Floratou, A., Minhas, U.F., Özcan, F.: SQL-on-Hadoop: full circle back to shared-nothing database architectures. Proc. VLDB Endowment **7**(12), 1295–1306 (2014)
8. Krishnamoorthy, M.S.: A note on some simplified NP-complete graph problems. ACM Sigact News **9**(3), 24–24 (1977)
9. Welsh powell algorithm. https://iq.opengenus.org/welsh-powell-algorithm/
10. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for owl knowledge base systems. Web Semant. Sci. Serv. Agents World Wide Web **3**(2–3), 158–182 (2005)
11. Chang, L., et al.: HAWQ: a massively parallel processing SQL engine in hadoop. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 1223–1234. ACM (2014)
12. Peng, P., Zou, L., Chen, L., Zhao, D.: Adaptive distributed RDF graph fragmentation and allocation based on query workload. IEEE Trans. Knowl. Data Eng. **31**(4), 670–685 (2018)
13. Papailiou, N., Tsoumakos, D., Konstantinou, I., Karras, P., Koziris, N.: H 2 RDF+: an efficient data management system for big RDF graphs. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of data, pp. 909–912. ACM (2014)
14. Schätzle, A., Przyjaciel-Zablocki, M., Neu, A., Lausen, G.: Sempala: interactive SPARQL query processing on hadoop. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 164–179. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_11
15. He, L., et al.: Stylus: a strongly-typed store for serving massive RDF data. Proc. VLDB Endowment **11**(2), 203–216 (2017)