RIBStore: Data Management of RDF triples with RDFS Inferences

Olivier Curé, David Célestin Faye, Guillaume Blin

RIBStore: Data Management of RDF triples with RDFS Inferences

Olivier Curé[1], David Faye[1,2], Guillaume Blin[1]

[1] Université Paris-Est, LIGM - UMR CNRS 8049, France
{ocure, gblin}@univ-mlv.fr
[2] Université Gaston Berger de Saint-Louis, LANI, Sénégal
dfaye@igm.univ-mlv.fr

Abstract. The vision of the Semantic Web is becoming a reality with billions of RDF triples being distributed over multiple queryable endpoints (e.g. Linked Data). Although there has been a body of work on RDF triples persistent storage, it seems that the problem of providing an efficient, in terms of query performance and data redundancy, inference enabled approach is still open. In this work, we take benefit of recent papers proposing a vertically-partition approach implemented on a column-oriented relational database management system and extend it with a novel approach enabling to represent less relations, thus requiring less joins in practical queries. This extension uses the fact that properties are first-class citizens in the RDF model to propose a particularly efficient solution when inferences are performed on property hierarchies. Another contribution of this paper is to propose a set of semantic query rewriting rules to improve query performance by reasoning over the ontology schema of the RDF triples. We provide an experimental evaluation on synthetic databases which emphasizes the relevance of our two contributions in efficient RDF triples storage.

1 Introduction

The vision of the Semantic Web, as proposed by [6], is becoming a reality with billions of RDF triples and hundreds of RDFS/OWL ontologies being distributed over multiple queryable endpoints (e.g. Linked data). These endpoints can generally be queried using the SPARQL query language [22] or in a programmatic manner via one of the many available RDF APIs [1]. Practically, these queries usually require some form of reasoning, a feature not natively supported by the current SPARQL W3C's recommendation. An approach generally encountered consists in performing inferences in an RDFS/OWL compliant reasoner, to use their results in order to generate a set of queries and to execute them over the appropriate data sets. This approach corresponds to the one we have designed in a medical application [12] where inferences need to be performed on chemical molecules in order to detect contra indications, side effects, etc. In Fig. 1, we present an extract of the information stored for the Ibuprofen molecule concerning several forms of contra indications. Moreover, our ontology contains a contra indication

property hierarchy which is defined as follows: (1) `diseaseContraIndication` $\sqsubseteq$ `contraIndication`, (2) `stateContraIndication` $\sqsubseteq$ `contraIndication`, (3) `moleculeContraIndication` $\sqsubseteq$ `contraIndication` with all `contraIndication` siblings being mutually disjoint. An interesting query over this data set would be to retrieve all objects contra indicated to the `Ibuprofen` molecule. That is, we want to retrieve all sub properties of `contraIndication` (namely {`molecule, state, disease`}`ContraIndication`) and to generate for each one a query that retrieves the objects where the subject is `Ibuprofen`. On the dataset of Fig. 1, this would yield a result set containing all entries of the last column.

Fig. 1. Extract of the contra indications for the `Ibuprofen` molecule

Subject	**Property**	**Object**
Ibuprofen	moleculeContraIndication	Ticlopidin
Ibuprofen	moleculeContraIndication	Clopidrogel
Ibuprofen	stateContraIndication	Breast feeding
Ibuprofen	stateContraIndication	Pregnant
Ibuprofen	diseaseContraIndication	Hypertensive heart

Another form of inference-based query would be to retrieve all objects contra indicated to all `Nonsteroidal Anti-Inflammatory Drugs` (e.g. `Ketoprofen`). Note that queries with such inference patterns can be required by an application as well as by some data quality or data exchange external tools.

Providing efficient performances to reasoning dependent queries is an important issue when the ontology and data sets are large (e.g. `OpenGalen` or `SNOMED`). We believe that to enable efficient response time to these queries, one has to give a special attention to these triples storage system. But `RDF` is basically a data model and its recommendation does not guide to a prefered storage solutions.

In [3], a novel approach to store `RDF` triples (*i.e.* subject, property and object, see Section 2 for more details) is presented, implemented in a system named `swStore` and evaluated. It consists in creating a relation containing only two columns (one for each the subject and the object of a triple) for each property in the ontology. Due to this relation structure, the implantation of this approach is particularly efficient on a column-oriented database system. Nevertheless, the authors of [3] never consider any forms of reasoning in the queries they evaluate on their solution. We consider that this is an important drawback of this approach since inferences is a key feature in applications dealing with `RDF` graphs. Moreover, [23] emphasized that this approach presents some limits when the set of ontology properties is large.

In this paper, we address both of these issues by proposing a solution that outperforms the approach of `swStore` when reasoning over property hierarchies is necessary. The performance improvement are due to the generation and maintenance of less relations than in `swStore`. The approach consists in creating a single property table for each property hierarchy in the ontology. Such a table contains three columns which correspond to standard `RDF` triples. Hence per-

formance of an important number of queries requiring inferences on property hierarchies are improved since less joins are needed.

In order to address the notion of RDFS inferences in our storage of RDF triples, we also propose a set of semantic query rewriting rules for the generation of SQL queries. These rules tackle all the inferences possible in RDFS and generate a single SQL query retrieving the correct answer set.

In this paper, we make the following contributions: (1) we extend the RDF triples storage solution of swStore by minimizing the number of relations proportionnally to the number of property hierarchies present in the RDFS ontology. This approach is particularly relevant when inferences are performed over property hierarchies, outperforming swStore by several orders of magnitude; (2) we propose a set of semantic query rewriting rules to enforce RDFS inferences when retrieving data from the underlying relational database; (3) we evaluate this new approach on both a row and column store against the swStore solution.

The remainder of this paper is organized as follows. In Section 2, we provide background knowledge of some of the key notions used in this work: namely RDF, RDFS and SPARQL. Section 3 presents related work relevant in the RDF triple storage. In Section 4, we detail our storage approach, named RDFS Inference-Based Store (RIBStore) as well as the set of query rewriting rules we are proposing. Section 5 compares our approach with swStore via an evaluation on synthetic data sets over both row and column oriented databases.

2 Preliminaries

In this section, we briefly introduce the background required for the rest of this paper on the following W3C recommendations: RDF model, the RDFS vocabulary language and SPARQL, a query language for RDF.

RDF is a logical data model consisting of triples of the form $\langle$s,p,o$\rangle$ where s, p and o are resp. called the subject, property and object of the triple. The signature of an RDF triple corresponds to $(\mathtt{U} \cup \mathtt{B}) \times \mathtt{U} \times (\mathtt{U} \cup \mathtt{B} \cup \mathtt{L})$ where U, B and L are possibly infinite sets of respectively URI resources, blank nodes (a form of existentially quantified variable) and RDF literals.

The RDFS vocabulary (henceforth RDFS) specifies a set of reserved words used to describe relationships between resources and properties. We are particularly interested in the following two groups of terms, and invite interested readers to study [8] for further details: (1) classes from which instances are created; (2) properties, *i.e.* rdfs:subClassOf, rdfs:subPropertyOf, rdfs:range, rdfs:domain and rdf:type which resp. specify a sub class (resp. sub property) relationship between classes (resp. properties), the range and domain of a property, and the type of an instance. We summarize in Fig. 2 the sound and complete set of rules for the entailment of RDF graphs with RDFS (see [15] for details).

SPARQL is a graph-matching query language for RDF. A query is composed of three parts: (i) a pattern matching part which takes the form of triples and includes several interesting features, e.g. filtering, (ii) solution modifiers like distinct, order, limit, etc. (iii) output of the query. Several works are tackling some

Fig. 2. RDFS inference rules

$$(1)\ \frac{(a\ rdf{:}type\ rdf{:}Property)}{a\ rdfs{:}subPropertyOf\ a} \qquad (2)\ \frac{(a\ rdfs{:}subPropertyOf\ b)(x\ a\ y)}{x\ b\ y}$$

$$(3)\ \frac{(a\ rdfs{:}subPropertyOf\ b)(b\ rdfs{:}subPropertyOf\ c)}{a\ rdfs{:}subPropertyOf\ c} \qquad (4)\ \frac{(a\ rdf{:}type\ rdf{:}class)}{(a\ rdfs{:}subClassOf\ a)}$$

$$(5)\ \frac{(a\ rdfs{:}subClassOf\ b)(b\ rdfs{:}subClassOf\ c)}{a\ rdfs{:}subClassOf\ c} \qquad (6)\ \frac{(a\ rdfs{:}subClassOf\ b)(x\ rdf{:}type\ a)}{x\ rdf{:}type\ b}$$

$$(7)\ \frac{(a\ rdfs{:}domain\ b)(x\ rdf{:}type\ a)}{x\ rdf{:}type\ b} \qquad (8)\ \frac{(a\ rdfs{:}range\ c)(x\ a\ y)}{y\ rdf{:}type\ c}$$

of the current limitations of SPARQL, namely the lack of inference support and absence of standard update mechanisms.

3 Related work

This section gives an overview of techniques for storing RDF data. There is a real need to efficiently store and retrieve RDF data as the number and scale of Semantic Web in real-word applications in use increase. The related work about RDF data management systems can be subdivided into two categories : the ones involving a mapping to a Relational DataBase Management System (RDBMS) and the ones that do not.

3.1 RDBMS based solutions of RDF data storage

A set of techniques have been proposed for storing RDF data in relational databases. Several research groups think that this is likely the best performing approach for their persistent data store, since a great amount of work has been done on making relational systems efficient, extremely scalable and robust. Efficient storage of RDF data has already been discussed in the literature with different physical organization techniques such as *triple table*, *property table* and the *vertical partitioning* approach, each one requiring sometimes prioritization of one performance metric to another.

Triple table.

The *triple-table* approach is perhaps the most straightforward mapping of RDF into an RDBMS. Each RDF statement of the form $(subject, property, object)$ is stored as a triple in one large table with a three-columns schema (i.e. a column for the subject, property and object resp.). Indexes are then added for each of the columns in oder to make joins less expensive. It has been used by Sesame[9], Jena[27], Oracle[10] and 3store[14].

However, since the collection of triples are stored in one single RDF table, the queries may be very slow to execute. Indeed when the number of triples scales, the RDF table may exceed size of memory. Additionally, simple *statement-based* queries can be satisfactorily processed by such systems, although they do not represent the most important way of querying RDF data. In the other hand, RDF triples store scales poorly because complex queries with multiple triple patterns require many self-joins over this single large table as pointed out in [27, 25, 18].

Property table.

The property table technique has been introduced later on for improving RDF data organization by allowing multiple triple patterns referencing the same subject to be retrieved without an expensive join. In this model, RDF tables are physically stored in a representation closer to traditional relational schemas in order to speed up the queries over the triple stores[26, 10]. In this approach, each named table includes a subject and several fixed properties. The main idea is to discover clusters of subjects often appearing with the same set of properties. A variant of the property table named *property-class table* uses the property called "type" of subjects to cluster similar sets of subjects together in the same table.

The immediate consequence is that self-joins on the subject column can be avoided. However, the property table technique has the drawback of generating many NULL values since, for a given cluster, not all properties will be defined for all subjects. This is due to the fact that RDF data may not be very structured. A second disadvantage of property table is that multi-valued attributes, that are furthermore frequent in RDF data, are hard to express. In a data model without a fixed schema like RDF, its common to seek for all defined properties of a given subject, which, in the property table approach, requires scanning all tables.

Note that, in this approach, adding properties requires also to add new tables; which is clearly a limitation for applications dealing with arbitrary RDF content. Thus the flexibility in schema is lost and this approach limits the benefits of using RDF. Moreover, queries with triples patterns that involve multiple property tables are still expensive because they may require many union clauses and joins to combine data from several tables and consequently complicate query translation and plan generation. In summary, property tables are poorly used because of their complexity and inability to handle multi-valued attributes.

Vertically partitioned table.

The vertical partitioning approach suggested in [3] is an alternative to the property table solution that speeds up queries over a triple store providing similar performance while being easier to implement. In this approach, the RDF data is vertically partitioned by using a fully decomposed storage model (DSM)[11]. Each triples table is divided into n two columns tables where n is the number of unique properties in the data. In each of these resulting tables, the first column contains the *subject* and the second column the *object* value of that subject.

The tables being sorted by subject, one has a way to use fast merge joins to reconstruct information about multiple properties for subsets of subjects. The vertically partitioned approach offers a support for multi-valued attributes. Indeed, if a subject has more than one object value for a given property, each distinct value is listed in a successive row in the table for that property. For a given query, only the properties involved in that query need to be read and no clustering algorithm is needed to divide the triples table into two-column tables.

Note that inserts can be slow in vertically partitioned tables since multiple tables need to be accessed for statement about the same subject.

In [3], the authors described how a column-oriented DBMS[24] (i.e., a DBMS designed especially for the vertically partitioned case, as opposed to a row-oriented DBMS, gaining benefits of compressibility [4] and performance [2]) can

be extended to implement the vertically partitioned approach. This is done by storing tables as collections of columns rather than collections of rows. The goal is to avoid reading entire row into memory from disk, like in row-oriented databases, if only a few attributes are accessed per query. Consequently, in column oriented databases only those columns relevant to a query will be read.

In [23], an independent evaluation of the techniques presented in [3], the authors pointed out potential scalability problems for the vertically partitioned approach when the number of properties in an RDF data-set is high. With a larger number of properties, the triple store solution manages to outperform the vertically partitioned approach.

3.2 Other RDF data storage approaches

Most of these approaches eschew the mapping to an RDBMS and focus instead on indexing techniques specific to RDF data model. They are motivated by the fact that using a traditional RDBMS for RDF data storage results in propagating RDBMS deficiencies such as inflexible schemas whereas avoiding these limitations is, arguably, one of the major reasons for adopting the RDF data model[18]. These proposals aim to be closer to the query model of the Semantic Web.

Graph based RDF data storage

These solutions are in general based on main-memory graph implementations and face scaling limitations. In the approaches presented in [7, 17], the RDF data is stored as graph. Nevertheless, the authors did not emphasize the scalability problems. Others similar approaches[19] use the *path-based* schemes to store the subgraphs in distinct tables of a relational database.

Multiple-index frameworks

The systems using an index structure are: YARS, Kowari, Hexastore, RDFJoin and RDFKB.

The YARS[16] system stores RDF data by using six B+ tree indices. It stores not only the subject, the property and the object, but also the context information about the provenance of the data as a *quad*. Each element of the quad is encoded in a dictionary. In each B+ tree, the key is a concatenation of the subject, predicate, object and context. The six indices constructed cover all the possible access patterns of quads in the form $\langle$s,p,o,c$\rangle$ where c is the context of the triple $\langle$s,p,o$\rangle$. This representation allows fast retrieval of all triple access patterns. Thus, it is also oriented towards simple statement-based queries and has limitations for efficient processing of more complex queries. The proposal sacrifices space and insertion speed for query performance since each triple is encoded in the dictionary six times.

The Kowari[5] system uses an approach similar to YARS. The RDF statements are also stored as quads like in YARS. However, Kowari uses an hybrid of AVL and B trees instead of B+ trees for multiple indexing.

Hexastore[25] takes also a similar approach to YARS. The framework is based on the idea of main-memory indexing RDF data in a multiple-index framework. The RDF data is indexed in six possible ways, one for each possible ordering of the three RDF elements. Two vectors are associated with each RDF element, one for

each of the other two RDF elements (e.g., [subject,property] and [subject,object]). Moreover, lists of the third RDF element are appended to the elements in these vectors. Hence, a sextuple indexing scheme is created. These indices materialize all possible orders of precedence of the three RDF elements. The representation is based on any order of significance of RDF resources and properties and can be seen as a combination of vertical partitioning and multiple indexing approaches. Hexastore provides efficient single triple pattern lookups, and also allows fast merge-joins for any pair of two triple patterns. However, space requirement of Hexastore is five time the space required for storing statement in a triples table. Hexastore favors query performance over insertion times passing over applications that require efficient statement insertion.

The RDFJoin[20] project provides several new features built on top of Hexastore. Hexastore is a main-memory solution whereas RDFJoin proposes a persistent database storage for these tables. The authors store all the third column tuples in a bit vector, and provide hash indexing based on the first two columns. This reduces storage space and memory usage and improves the performance of both joins and lookups.

RDFKB[21] (Resource Description Framework Knowledge Base) is a relational database system for RDF datasets supporting inference and knowledge management. The solution is implemented and tested using column store and the RDFJoin[20] technology and supports inference at data storage time rather than as part of query processing. All known inference rules are applied to the dataset to determine all possible knowledge. The inferred knowledge is stored in the dataset. The authors make the choice to store redundant information. At query execution time, there is information about which knowledge relates to the query, and this can be used to limit the scope of the inference search. Queries against inferred data are simplified, and performance is increased. However, inferring all possible knowledge may be very expensive and the performance penalty can be high as the vocabulary is increased.

4 The RIBStore approach

As previously mentioned, we propose a new approach - namely RIBStore - which provides two main contributions: storing less relations (thus requiring less joins in practical queries) and a set of semantic query rewriting rules improving query performance. We consider that the applications which are using our approach express their queries in SPARQL, which are then translated into SQL queries and executed over a relational database. The framework we are providing enables, through the use of an API, to express inference requests within a SPARQL query. This enables end-users to specify that all properties (resp. concepts) of a hierarchy are needed in a query. Note that such a mechanism is currently not specified in the SPARQL recommendation.

4.1 Storage approach

The storage approach extends `swStore` by adding a third column to each property belonging to a property hierarchy. This new column contains property names. The reduction of the number of property tables has a big impact on the performance of queries requiring joins over properties of same property hierarchy. This is typically the case when one wants to reason over the triples of a property hierarchy. In `swStore` and `RIBStore`, a two columns relation is created for properties of the `RDF` and `RDFS` vocabularies, *e.g.* rdf:type, rdfs:subClassOf. And for all other properties of the ontology, we apply the following approach: (1) if a property is not part of a property hierarchy, generate a relation as in `swStore`. That is a two columns relation with subject and object attributes and the property name as relation name; (2) otherwise, for all properties in a property hierarchy, generate a single relation with the name of the top property of the hierarchy as relation name and a three columns pattern (subject, object and property). Each tuple of this relation contains the name of the property of the corresponding triple in the property column.

Example 1 Fig. 3 proposes an example with a property hierarchy (Fig. 3a) and a small data sets (Fig. 3b). With `swStore`, the triples would be distributed over 5 different relations as displayed in Fig. 3c. Comparatively, in `RIBStore` a single relation named after the top property of the hierarchy is created and would correspond exactly to Fig. 3b. Thus, we consider an ontology consisting of `n` property hierarchies with an average of `k` properties in each hierarchy. The `RIBStore` approach will store `n` $\times$ `k` less relations than a `swStore` approach.

Fig. 3. Storage comparison of swStore and RIBStore

(a) `pa` property hierarchy:
`pb` $\sqsubseteq$ `pa`,
`pc` $\sqsubseteq$ `pa`,
`pd` $\sqsubseteq$ `pc`,
`pe` $\sqsubseteq$ `pc`

(b) triple extract

sub.	prop.	obj.
a	pa	b
c	pc	d
e	pb	f
g	pe	h

(c) `swStore` relations

sub.	obj.
a	b

`pa` relation

sub.	obj.
g	h

`pb` relation

sub.	obj.
c	d

`pc` relation

sub.	obj.

`pd` relation (empty)

sub.	obj.
e	f

`pe` relation

We now consider the following query: one wants to retrieve all objects involved in a triple with a property of the `pa` hierarchy. With `swStore`'s physical design, the following query is needed:

```
SELECT object FROM pa UNION (SELECT object FROM pb UNION (SELECT
object FROM pc UNION (SELECT object FROM pd UNION (SELECT object FROM
pe))));
```

while the same query is answered far more efficiently in `RIBStore` with:

```
SELECT object FROM pa;
```

4.2 Inference-based query rewriting

The second contribution of this work corresponds to a set of semantic query rewriting rules. The semantic aspect of this rewriting is provided by a thorough usage of the RDFS inferences described in Section 2. The goals of this approach are (i) to detect, relying exclusively on ontology inferences, if the answer set of a query is empty or not and (ii) to optimize a given query via a semantic rewriting of some SQL queries. The rules can be decomposed into two sets: (1) A set of rules, denoted `subsume`, dealing with concept and property subsumptions which are being dealt with rules (1) to (6) of Fig. 2; (2) A set of rules, denoted `propertyCheck`, dealing with the domain and range of a given property. They are inferred by rules (7) and (8) from Fig. 2.

The rules processed by the `subsume` procedure are using the RDFS inference rules to compute all the sub concepts (resp. properties) of a given concept (resp. property). All RDFS APIs propose efficient implementation of rules (1) to (6) of Fig. 2. The query studied in Example 1 was already using the `subsume` procedure.

Example 2 Consider that the range of the property `pb` in Example 1 is of type `ClassA` which is the top concept in the following concept hierarchy: `ClassC` $\sqsubseteq$ `ClassA`, `ClassB` $\sqsubseteq$ `ClassA` and `ClassC` $\sqsubseteq$ `ClassB`. That is `ClassA` has two sub concepts which are disjoint. Consider a query asking for all subjects and objects of triples where `pb` is the property and all subjects belong to the `ClassA` hierarchy. Using `subsume`, the query can be translated in the following SQL query:

```
SELECT subject, object FROM pb, type WHERE type.subject =
pb.subject AND type.object IN ('ClassA','ClassB', 'ClassC');
```

Thus this approach enables to generate a singe SQL query whatever the size of the concept hierarchy. Note that it also applies to property hierarchies.

The rules of `propertyCheck` are being processed as follows: first the SPARQL query is parsed and for each property explicitly mentioned in the query with a typed (`rdf:type`) subject or object, we store the property name and types of the subject and/or object. Then for each subject (resp. object) in the structure, we search if there is a direct or indirect (via subsumptions) correspondence with the type of the `rdf:domain` (resp. `rdf:range`) defined in the ontology for this property.

Example 3 Consider the triples defined in Fig. 1 with the following ontology axioms: (1) `rdf:range` of `diseaseContraIndication` is an instance of the `Disease` concept, (2) `Disease` $\sqsubseteq$ `Top`, (3) `Molecule` $\sqsubseteq$ `Top` and (4) `Disease` $\sqsubseteq \neg$ `Molecule`. Intuitively, axioms (2) to (4) state respectively that the `Disease` concept is sub concept ot the `Top` concept, identically for the `Molecule` concept and finally, the `Disease` and `Molecule` concepts are disjoint. Consider the following SPARQL query:

```
SELECT ?s ?o WHERE {?s :diseaseContraIndication ?o.
?o rdf:type :Molecule.}
```

which asks for subjects and objects involved in triples where the property is `diseaseContraIndication` and the object has a type `Molecule`. Clearly the answer set to this query is empty since the `rdf:domain` of the property can not be a `Molecule` is this ontology.

Example 4 Consider the context of Example 3 with the following query:

```
SELECT ?s ?o WHERE {?s :diseaseContraIndication ?o.
?o rdf:type :Disease.}
```

The query is satisfiable since there is model where its answer set is not empty. Anyhow, the query can be optimized. In fact, it is not necessary to check the `rdf:type` of the object because it corresponds exactly to the one defined as `rdf:range` in the ontology. Thus this query is rewritten in:

```
SELECT ?s ?o WHERE {?s :diseaseContraIndication ?o.}
```

which once translated into `SQL` does not require any join and will thus perform far more efficiently than the orginal query.

5 Evaluation

We now describe an experimental study that stresses the effectiveness and relevancy of the techniques presented in Section 4. First, we provide details of the experiment settings, *i.e.* data sets, queries, benchmark system, database management systems and reasoner. Then results are represented and analyzed in a systematic way.

5.1 Experimental settings

All our experiments have been conducted on four synthetic databases. They all have been generated from the Lehigh University Benchmark (LUBM) [13] which has been developed to facilitate the evaluation of Semantic Web repositories in a standard and systematic way. The `RDF` data sets generated with LUBM all commit to a single realistic ontology dealing with the university domain. This ontology is composed of 43 concepts, 25 object properties and 7 data type properties. Since we are interested in inferences using concept and property hierarchies, our queries will take advantage of the 36 subclass and 5 subproperty axioms. Some of our queries imply to infer on subclass axioms. For this purpose, we have selected the hierarchy of the $Person$ class from which we present and extract: $AssociateProfessor \sqsubseteq Professor \sqsubseteq Faculty \sqsubseteq Employee \sqsubseteq Person$, $AdministrativeStaff \sqsubseteq Employee$ and $Faculty \sqsubseteq \neg AdministrativeStaff$. The sub property axioms we are using the most in our evaluation (queries 1 to 6) concern the membership hierarchy which is classified as follows : $headOf \sqsubseteq worksFor \sqsubseteq memberOf$. Surprisingly, these 3 object properties are underspecified since no domain and range are provided. Since we also want to evaluate our query rewriting technique, queries 7 to 9 are using the $teacherOf$ property because its domain and range are clearly defined.

This ontology serves as the schema underlying the 4 data sets we have created. This is an important requisite for our evaluation since our set of queries will be executed on all data sets in order to provide information on scalability issues. Table 1 summarizes the main characteristics of these data sets in terms of overall number of triples, number of concept and property instances.

Table 1. Synthetic datasets

DB name	Universities	Concept instances	Property instances	Triples
lubm1	1	15195	60859	100868
lubm2	2	62848	251252	236336
lubm5	5	114535	456137	643435
lubm10	10	263427	1052895	1296940

The `RDF` data sets are later translated into the different physical models we would like to evaluate. They are decomposed into two main approaches: the solution of [3] – henceforth refered as **swStore** – and the solution we have proposed in this paper: **RIBStore**. In order to emphasize the efficiency of our solution in situations needing reasoning, we had to test these settings in a similar context as [3]. More precisely, each conceptual solution needs to be evaluated on a row store and a column store RDBMS. This yields the four following approaches: `swStore` resp. on a row (**swRStore**) and column (**swCStore**) store and `RIBStore` resp. on a row (**RIBRStore**) and column (**RIBCStore**) store. Hence a total of 16 databases are generated (each data set is implemented on each physical approach).

We have selected postgreSQL and MonetDB as the RDBMS resp. for the row-oriented and the column-oriented databases. We retained MonetDB instead of C-store (the column store used for evaluation in [3]) essentially due to (1) the lack of maintenance of the latter one, (2) the open-source licence of MonetDB and (3) the fact that MonetDB is considered state of the art in column-oriented databases. The tests were run on MonetDB server version 5 and postgreSQL version 8.3.1. The benchmarking system is an Intel Core 2 Duo T7700 2.4 GHz operated by a Linux Ubuntu 9.10, with 2Gbytes of memory, 4MB L2 cache and one disk of 160 Gbyte spinning at 7200rpm. The disk can read cold data at a rate of approximatively 55MB/sec. For the `swRStore`, there is a clustered B+ tree index on the subject and an unclustered B+ tree on the subject. Similarly, for the `RIBRStore`, a clustered B+ tree index is created on the property column and an unclustered B+ tree on the subject. As noted in [23] MonetDB does not include user defined indices. Hence we relied on the ordering of the data on property, subject and object in both `swCStore` and `RIBCStore`.

We have designed 9 queries to evaluate our approach. They can be decomposed into 2 sets depending on their need for inferences.

The set of queries not requiring any form of reasoning is composed of:

- Query 1 (Q1): retrieves the subject of triples where the property is $memberOf$.
- Query 2 (Q2): retrieves the subject of triples where the property is $headOf$. This query is interesting since the distributions of $memberOf$ and $headOf$ axioms in all data sets is not equiprobable (*i.e.* there is an average factor of 500 between the number of instances of these two properties, *e.g.* 7490 $memberOf$ and 15 $headOf$ axioms in lubm1).
- Query 3 (Q3): provides a result set with subjects and objects involved in triples where the property is $worksFor$ and the subject has a type concept

equal to $AssociateProfessor$. Thus this query requires a join between the *type* relation and the $memberOf$ (resp. $worksFor$) relation in `RIBStore` (resp. `swStore`).

The set of queries requiring inferences is composed of the following 6 queries:

- Query 4 (Q4): retrieves all subjects involved in triples where the property is one of the properties of the $memberOf$ property, i.e. $memberOf$, $headOf$ and $worksFor$. This query is an extension of Q1 requiring reasoning.
- Query 5 (Q5): extends Q3 with reasoning over the concept hierarchy of the $Professor$ concept. That is, all subjects and objects involved in triples where the property is $worksFor$ and the type of subject is one of the 6 distinct concepts derived from $Professor$.
- Query 6 (Q6): further extends Q5 by considering that properties of the selected triples are in the $memberOf$ property hierarchy. This means that reasoning at the property and concept levels are required.
- Query 7 (Q7): selects the subject and the object in triples where the property is $teacherOf$ and subject is of type $AdministrativeStaff$. This query returns an empty answer set since the domain of $teacherOf$ is the $Faculty$ concept which is disjoint with $AdministrativeStaff$. This query enables to test our rewriting query approach which is defined declaratively in our framework but it could also be computed via a query written in `SPARQL`.
- Query 8 (Q8): selects the subject and the object in triples where the property is $teacherOf$ and subject is of type $Faculty$. This query requires a join.
- Query 9 (Q9): has the same purpose as Q8 but exploits one of our rewriting rules to improve the performances of Q8. In fact, the join in Q8 is not necessary if one knows that the domain of $teacherOf$ is the concept $Faculty$.

Finally, reasoning operations assume that the LUBM ontology is stored in main-memory and are performed using the Jena framework [1].

5.2 Experimental results

The analysis of figures 4, 5 and 6 of the queries not requiring inferences (Q1, Q2 and Q3) confirm the results highlighted in [3] and [23]; namely that the column store outperforms the row one. Unsurprisingly, the `swStore` approaches provide better query execution performances than `RIBStore` counterparts. This is easily comprehensible since for `swStore`, a simple scan of the tuples of the *memberOf* (Q1) or *headOf* (Q2) relations are sufficient while `RIBStore` requires a selection of tuples according to the values of the property column. These results are confirmed even in the presence of a join(Q3).

The remaining queries imply a form of reasoning and emphasize the effective approach of `RIBStore`. For instance, Fig. 7 shows the performances of query Q4 and clearly demonstrates the efficiency of `RIBStore` over `swStore`. Even the row oriented `RIBStore` outperforms the column oriented `swStore`. This is due to the presence of `UNION SQL` operators in the queries executed on the `swStore` while `RIBStore` only requires a complete scan of the tuples of one relation.

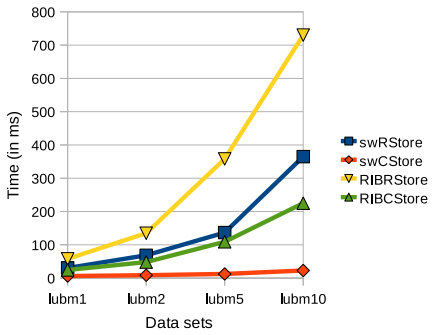

Fig. 4. Performance results for Q1

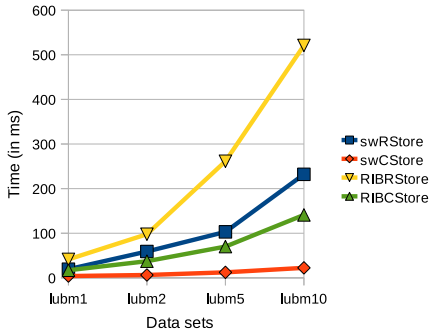

Fig. 5. Performance results for Q2

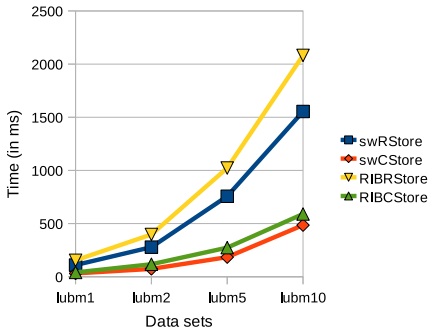

Fig. 6. Performance results for Q3

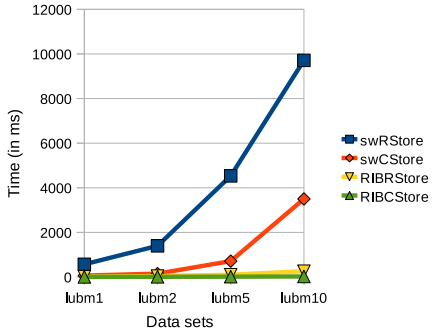

Fig. 7. Performance results for Q4

The conclusions of the analysis of Q1/Q2 and Q4 are confirmed even in the presence of reasoning over the concept hierarchy. This is not surprising since concept subsumption does not impact performance in `swStore` and `RIBStore`.

Finally, queries Q7, Q8 and Q9 emphasize the importance of reasoning over the ontology before executing queries over any of the store solutions. Fig. 10 displays the duration times for all databases, ranging from approximately 42ms (column store with 1 university) to 1450 ms (row store with 10 universities). This can be considered rather long to propose empty answer set since, according to the ontology, the query is incoherent. Comparatively, we have implemented a simple generic method which scans the query and checks if the domain and/or range of the selected property matches the concept introduced in the query as a subject or object. This method executes in an average time of 1ms and only depends on the ontology schema. Hence, a system implemented on top of an `RDFS` compliant reasoner is able to determine almost instantly if the answer set is empty. Moreover, it could also provide some explanations concerning the lack of tuples in the answer. We believe that such optimization are quite useful especially when end-users are not well aware of the details of the ontology.

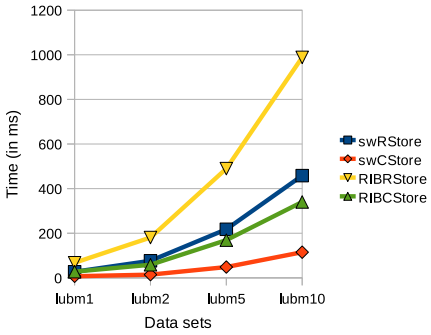

Fig. 8. Performance results for Q5

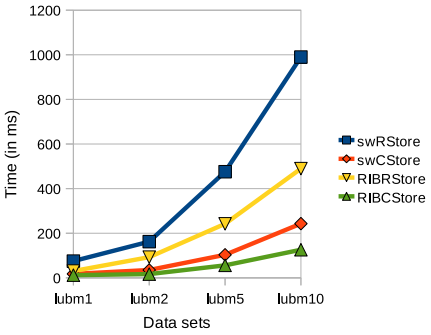

Fig. 9. Performance results for Q6

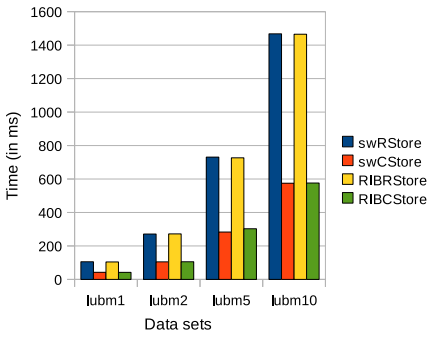

Fig. 10. Performance results for Q7

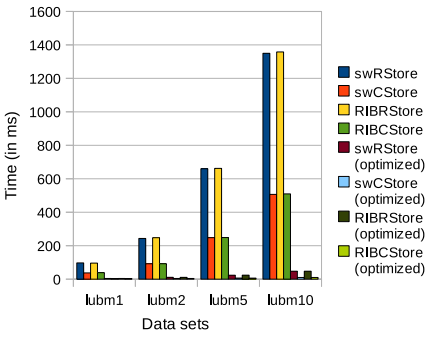

Fig. 11. Comparison of performance results of Q8 and Q9

The performance results of Q8 and Q9 are provided together in Fig. 11 in order to highlight their comparisons. The purpose of Q8 and Q9 is to emphasize the importance of analyzing property domain and range in a property table approach. The execution of Q8 does not perform any optimizations while Q9 checks that the concept *Faculty* is the domain of the *teacherOf* property and hence a join to the `rdf:type` relation is not necessary.

6 Conclusion

The first contribution of this paper is to propose a set of semantic query rewriting rules to support `RDFS` inferences when one wants to query `RDF` triples. This contribution adapts to the vertically partitioned approach of [3] by enabling domain, range, sub class and sub property inferences. In a second contribution, we extended the vertically partitioned approach with a third column corresponding to a property in the presence of property hierarchies. This extension was motivated by the fact that properties are first-class citizen in `RDF` and by the

interesting performances obtained when inferences are required on these hierarchies. Our new approach retains all the interesting properties presented in [3], namely support for multi-valued attributes and heterogeneous records, only properties accessed by a query are read, fewer unions and fast joins. Moreover, by storing less relations (one for each property hierarchy instead of the size of the property hierarchy), even less union and joins are needed. This is particularly valuable in many practical queries and when data quality operations are performed over the triple store.

In future works, we would like to extend our query rewriting rules to ontologies expressed in `OWL` and we would also like to consider other forms of storage layers outside of the field of relational databases.

References

1. Jena - a semantic web framework for java. `http://jena.sourceforge.net`.
2. Abadi, D.J., Myers, D.S., DeWitt, D.J., Samuel, R.M. : Materialization Strategies in a Column-Oriented DBMS. ICDE'07, 466-475, 2007
3. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K. : Scalable semantic web data management using vertical partitioning. VLDB '07, 411-422, 2007
4. Abadi, D.J., Madden, S., Ferreira, M. : Integrating compression and execution in column-oriented database systems. SIGMOD '06, 671–682, 2006.
5. Adams, T., Gearon, P., Wood, D. : Kowari: A Platform for Semantic Web Storage and Analysis. XTech'05, 2005.
6. Berners-Lee, T., Hendler, J., Lassila, O. : The Semantic Web. Scientific American, May 2001. Vol. 284, Number 5
7. Bönström, V., Hinze, A., Schweppe, H. : Storing RDF as a Graph. LA-WEB '03
8. Brickley, D., Guha, R.V.: RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation (Feb. 2004) `http://www.w3.org/TR/rdf-schema/`
9. Broekstra, J., Kampman, A., Van Harmelen, F. : Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. ISWC'02. 54–68 2002
10. Chong, E.I., Das, S., Eadon, G., Srinivasan, J. : An efficient SQL-based RDF querying scheme. VLDB'05. 1216–1227, 2005
11. Copeland, G.P., Khoshafian, S.N. : A decomposition storage model. SIGMOD'85, 268–279, 1985
12. Curé, O.: Semi-automatic Data Migration in a Self-medication Knowledge-based System. Wissensmanagement'05, 323-329
13. Guo Y., Pan Z., Heflin J. : LUBM: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005
14. Harris, S., Gibbins, N. : 3Store Efficient bulk RDF storage. PSSS'03, 1–20, 2003
15. Hayes, P.: RDF Semantics (Feb. 2004). `http://www.w3.org/TR/rdf-mt/`
16. Harth, A., Decker, S. : Optimized Index Structures for Querying RDF from the Web. LA-WEB '05, 71–80, 2005
17. Hayes, J., Gutierrez C. : Bipartite Graphs as intermediate model for RDF. ISWC'04, 47-61, 2004
18. Kolas, D., Emmons, I., Dean, M. : Efficient Linked-list RDF Indexing in Parliament. SSWS'09, 17-32, 2009
19. Matono, A., Amagasa, T., Yoshikawa, M., Uemura, S. : A path-based relational RDF database. ADC'05, 95-103, 2005

20. McGlothlin, J., Khan, L. : RDFKB: efficient support for RDF inference queries and knowledge management. IDEAS'09, 259-266, 2009
21. McGlothlin, J., Khan, L. : RDFJoin: A Scalable of Data Model for Persistence and Efficient Querying of RDF Datasets. Technical Report UTDCS-08-09.
22. Prud'hommeaux, E., Seaborn A.: SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`
23. Sidirourgos, L., Goncalves, R., Kersten, M., Nes, N., Manegold, S. :Column-store support for RDF data management: not all swans are white. VLDB'08, 1553-1563
24. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E.J., O'Neil, P., Rasin, A., Tran, N., Zdonik, S. : C-store: a column-oriented DBMS. VLDB'05, 553-564, 2005
25. Weiss, C., Karras, P., Bernstein, A. : Hexastore : sextuple indexing for semantic web data management. VLDB'08, 1008-1019, 2008
26. Wilkinson, K. : Jena property table implementation. SSWS'06, 2006
27. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D. : Efficient RDF Storage and Retrieval in Jena2. SWDB'03, 131-150, 2003