RDF triples management in roStore

David Faye, Olivier Curé, Guillaume Blin, Cheikh Thiam

RDF triples management in `roStore`

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

[1] LANI, Université Gaston Berger de Saint-Louis, Sénégal
{david-celestin.faye,cheikh.thiam}@ugb.edu.sn

[2] Université Paris-Est, Marne-la-Vallée, LIGM - UMR CNRS 8049, France
{ocure, gblin}@univ-mlv.fr

Résumé : This paper tackles issues encountered in storing and querying services dealing with information described with Semantic Web languages, e.g. OWL and RDF(S). Our work considers RDF triples stored in relational databases. We assume that depending on the applications and queries asked to RDF triple stores, different partitioning approaches can be considered : either storing all triples in a single relation or using a vertical partitioning where each property is associated to a given relation. We believe that several solutions lie in between these two approaches and we have already proposed `roStore` has one of them. It consists of an ontology-guided, column-oriented approach particularly efficient for ontologies containing property hierarchies. In a previous paper, we have emphasized that this approach is efficient for queries retrieving information. Due to the adoption of a column-oriented relational approach, an obvious question is : How does it perform on update operations ? The main contribution of this paper is to reply to this question through an evaluation on knowledge bases generated from LUBM. Moreover, our previous work is also extended with (i) a solution to adapt the `roStore` schema when the underlying ontology is modified and (ii) a formalization of an inference-based query translation from SPARQL to SQL for `roStore`.

Mots-clés : Ontology, Semantic Web, RDF triples, Reasoning, Persistent storage

1. Introduction

Scalability issues are a main concern for most application designers. This also applies to Web applications where figures of several tera bytes of new data per day are not uncommon (e.g. in scientific and social applications). Some of these applications describe part of their data using a Semantic Web language, i.e. OWL and/or RDF(S). This aspect forces Semantic Web application designers to think about scalable storage solutions for knowledge bases. In this work, we only consider solutions using Relational DataBase Manage-

ment Systems (RDBMS) to store RDF triples. In a previous paper (Curé & Faye 2010), we argued that a spectrum of solutions are possible in this area and that the motivation of picking one solution instead of another is motivated by the kind of applications and queries asked to the triple store. Moreover, we identified a particular approach, named ontology-guided, which proposes an information partitioning based on the structure of the ontology. Our solution, namely `roStore`, adopts a property-based partitioning which we have shown to be efficient for data retrieving queries (e.g. SELECT queries).

The underlying physical model of `roStore` consists in a column-oriented RDBMS, i.e. storing tables as collections of columns rather than collections of rows, which is known to be more I/O efficient for read-only queries. An obvious question is : how `roStore` performs on update operations, i.e. insertion, modification and deletion of triples of the knowledge base ? This also tackles the issue of updates at the ontology level and how it impacts the relational database schema. This paper replies to these questions through respectively an evaluation on some LUBM knowledge bases with an adapted set of update queries and a presentation of a set of rules enabling an automatic transformation of the underlying relational database schema from modifications of the ontology. Finally, our (Curé & Faye 2010) paper is also extended with a SPARQL extension proposition supporting an inference-based approach of SPARQL to SQL transformations.

This paper is organized as follows. In Section 2, we enrich the related work proposed in (Curé & Faye 2010) with considerations on update operations performed on persistent RDF storage. Section 3 presents the `roStore` approach and provides details about the impact of RDF Schema evolution on the relational schema. In Section 4, we detail a set of extensions provided to the SPARQL query language to integrate inferences at the data management level. Section 5 proposes an evaluation on performing updates on `roStore`. Finally, Section 6 concludes the paper and presents some future work.

2. Related work

In this related work section, we extend the one proposed in (Curé & Faye 2010) by concentrating on update operations. That is we consider the support of modifications at the extensional level of the knowledge base. Due to the large amount of solutions for the storage of RDF triples and to space limitations, we concentrate on those solutions prevailing in a database context and accepting SPARQL queries. We will provide a particular attention to the

notion of indexes since it partly motivates the efficiency of update queries.

The idea of the Multiple Access Pattern (MAP) approach is the construction of indices that cover all the possible access patterns of the form ⟨s,p,o,⟩ where stands s for subjects , p predicates, and o for objects. All the three positions of a triple are indexed for some permutation of s, p, and o. The indexation is done by using up to six separate B+trees, corresponding to the six possible orderings, i.e. spo, sop, pso, pos, osp, ops. Among others systems using this technique we can cite YARS (Harth & Decker 2005), Virtuoso (Erling & Mikhailov 2007) and RDF-3X (Neumann & Weikum 2008).

The YARS system combines methods from databases and information retrieval to allow for better query answering performance over RDF data. It stores RDF data persistently by using six B+ tree indices. It not only stores the subject, the predicate and the object, but also the context information, denoted c, about the origin of the data. Each element of the corresponding quad (i.e. 4-uplet) is encoded in a dictionary storing mappings from literals and URIs to object IDs (OIDs – stored as number identifiers for compactness). To speed up keyword queries, the lexicon keeps an inverted index on string literals to allow fast full-text searches. 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 ⟨s, p, o, c⟩. This representation allows fast retrieval for 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, to retrieve any access patterns with a single index lookup, each triple is encoded in the dictionary six times, in different sorting order. Note that inference is not supported.

The commercial system Virtuoso (Erling & Mikhailov 2007) stores quads combining a graph to each triple ⟨s, p, o⟩.Thus, it conceptually stores the quads in a triples table expanded by one column. The columns are g for graph, and the standard s, p, o triple. While technically rooted in an RDBMS, it closely follows the model of YARS but with fewer indices. The quads are stored in two covering indices, g, s, p, o and o, g, p, s, where the URI's are dictionary encoded. Several further optimizations are added, including bitmap indexing. In this approach, the use of fewer indices tips the balance slightly towards update query operation performances but it still performs efficiently for retrieving queries.

RDF-3X (Neumann & Weikum 2008) is an RDF storage system with advanced indexes and query optimization that eliminates the need of physi-

cal database design by the use of exhaustive indexes for all permutations of subject-property-object triples. Neumann *et al.* use a potentialy huge triples table, with their own storage implementation underneath (as opposed to using an RDBMS). They overcome the problem of expensive self-joins by creating a suitable set of indexes. All the triples are stored in a compressed clustered B+ tree. The triples are sorted lexicographically in the B+ tree. The triple store is compressed by replacing long string literals in the triples IDs using a *mapping dictionary*. The system supports both individual update operations and entire batches updates.

`Hexastore` (Weiss & Karras 2008) is based on the idea of main-memory indexing of `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 by individual columns. The representation is based on any order of significance of `RDF` resources and properties and can be seen as a combination of vertical partitioning (Abadi & Marcus 2007) and multiple indexing approaches (Harth & Decker 2005). Two vectors are associated with each `RDF` element, one for each of the others 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 schema is created. As Weiss *et al.* point out in (Weiss & Karras 2008), the values for `o` in `pso` and `spo` are the same. So in reality, even though six tables are created only five copies of the data are really computed, since the object columns are duplicated. To limit the amount of storage needed for the URIs, Hexastore uses the typical dictionary encoding of the URIs and the literals, i.e. every URI and literal is assigned a unique numerical identifier. 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 time passing over applications that require efficient statement insertion. Updates and insertions operations affect all six indices, hence can be slow. Note that `Hexastore` does not provide inference support. Recently, in (Weiss & Karras 2008), Weiss *et al.* proposed an on-disk index structure/storage layout so that `Hexastore` performance advantages can be preserved. Additionally to their experimental evaluations, they show empirically that, in the context of `RDF` storage, their vector storage schema provides significantly lower data retrieval times compared to B trees.

The `RDFJoin` (McGlothlin & Khan 2009) project provides several new features built on top of previous cutting edge research including vertical parti-

tioning (Abadi & Marcus 2007) and sextuple indexing (Weiss & Karras 2008). `RDFJoin` proposes a persistent column-store database storage for these tables with the primary goal to reduce the need and cost of joins and unions in query implementations. Indeed, it also use the six possible indexes on ⟨`s, p, o`⟩ using three tables : `ps-o`, `so-p` and `po-s`. These tables are indexed on both the first two columns so they provide all possible six indexes, while insuring that only one copy of the third column is stored. By keeping three separate triples tables and normalizing the identification numbers, RDFJoin allows subject-object and object-object joins to be implemented as merge joins as well. `RDFJoin` uses conversion tables closely matching the dictionary encoding of the vertical partitioning. All the third column tuples are sotred in a bit vector, and hash indexing based on the first two columns is provided. This reduces space and memory usage and improves the performance of both joins and lookups. For example, the `ps-o` table has columns $Property$, $SubjectID$ and $ObjectBitVector$ where $ObjectBitVector$ is a bit vector with the bits corresponding to all the object ID that appears in a triple with this property and subject. This also applies for the `so-p` and the `po-s` tables. Thus, all of the `RDF` triples in the dataset can be rendered from any of these tables. Additionally, execution of subject-subject, subject-object and object-object joins are done and stored as binary vectors into tables called *join tables*. This task is performed one time for any `RDF` dataset during the preprocessing stage to avoid overhead. Then, the results are stored in the relational database where they are quickly accessible. Indeed, `RDFJoin` stores much of its data as binary vectors and implements joins and conditions as binary set operations. This implementation provides significant performance improvement over storing each triple as a unique tuple. Let us remark that `RDFJoin` does support insertion of new `RDF` triples, but does not allow direct updates or deletions of triples in the database. Moreover, there is no suppport for inference in `RDFJoin`.

The storage and indexing strategies used by each proposal may depend if the tool is concerned with query performance of adding or updating knowledge to the database. Considering update queries, we note the following limitations ; (i) information about a piece of data can appear in multiple locations, possibly spanning several different data structures. ,(ii) redundant storage, e.g. each B+tree in a MAP scheme contains a separate copy of essentially the same set of data , (iii) locating all triples related to some data requires lookups in three different data structures : increased query processing costs (e.g. performing a join on an atom can require multiple independent index lookups).

Note that none of these solutions uses inference for update queries.

3. roStore

3.1. Approach overview

In order to grasp this paper's contributions, we need to introduce the `roStore` approach. Nevertheless, we invite the interested reader to read (Curé & Faye 2010) to get more details and a motivation of its building blocks.

The `roStore` approach derives from the vertically partitioned one and extends it by clustering into a single table data related to a given *top-property* of a property hierarchy. Starting from a property hierarchy, we consider that a *predicate* is a top-property if it is not an `rdf:subPropertyOf` of another *predicate*. Then, for each top-property P^T, a three-columns table is created by (1) merging all the two-columns tables corresponding to *predicates* being `rdf:subPropertyOf` P^T and (2) adding a third column indicating from which *predicate* the entry *(subject, object)* was retrieved. This approach can be compared to the standard approaches proposed in Example 1 and Figure 1.

In `roStore`, any *predicate* not corresponding to an `rdf:subPropertyOf` of a top-property will still be stored in a two-columns table. This implies an insignificant expense of the space complexity of this novel approach (particularly if entries are encoded using a dictionary). Moreover, in case of a hierarchy not in the shape of a directed acyclic graph (i.e. DAG, which should be rarely encountered), any *predicate* being part of a cycle will be stored in a two-columns table (since we would not be able to define a specific top-property among them). Considering that this top-property based approach seems to be a natural approach, one may, depending on the topology of the hierarchy, define other physical organizations inducing better performance for specific cases. The major impact of merging some tables is to obtain better performance of queries requiring joins over *predicates* belonging to the same "sub-hierarchy" of the property hierarchy. This is typically the case when one wants to retrieve all the triples associated to a set of *predicates* in the same property hierarchy. In the following, we will denote by `vpStore` (resp. `roStore`) the vertically partitioned (resp. our) approach.

Example 1 : In this example, we use the LUBM ontology (Guo & Pan & al 2005) which has been developed to facilitate the evaluation of Semantic Web repositories in a standard and systematic way Consider the extract of Figure 1b of a LUBM dataset defined over the given property hierarchy of Figure 1a.

With `vpStore`, the triples would be distributed over three different tables as displayed in Figure 1c, d and e. Comparatively, in `roStore`, one obtains only one table : a single relation containing subject, object and property attributes, named after the top-property `memberOf` and storing all triples.

(a) property hierarchy : `headOf` $\sqsubseteq$ `worksFor` $\sqsubseteq$ `memberOf`

sub	prop	obj
a	headOf	b
c	worksFor	d
e	worksFor	f

(b) RDF triples

sub	obj
a	b

(c) headOf

sub	obj
c	d
e	f

(d) worksFor

sub	obj

(e) memberOf

FIGURE 1: Storage comparison of `vpStore` (sub, prop and obj denote resp. subject, property and object of a triple.

Thus, if we consider an ontology consisting of `n` (e.g. 2 in our example) property hierarchies with an average of `k` (e.g. 3 in our example) properties in each hierarchy, the `roStore` approach will store `k` times less tables than a `vpStore` approach. Moreover, with this approach it is very unlikely to generate tables with no tuples (e.g. `memberOf` with `vpStore` in Example 1). The set of tuples stored is the same than in `vpStore` and only their distribution over database tables is modified (*i.e.* physical organization).

We now consider the following query : one wants to retrieve all *objects* involved in a triple with a *predicate* of the `memberOf` hierarchy. Considering `vpStore`'s physical design, the following `SQL` query is needed :

```
SELECT object FROM memberOf UNION (SELECT object
FROM worksFor UNION (SELECT object FROM headOf));
```

while the same query is answered far more efficiently considering `roStore`'s physical design with : `SELECT object FROM memberOf;`

Such a simple example already emphasizes the improvement one can get from using a physical organization such as `roStore` over `vpStore` when property hierarchies are present in the ontology, e.g. the OpenGalen ontology contains a property hierarchy of depth 6.

3.2. RDF Schema evolution

The `roStore` system handles the creation of database relations given an ontology represented in OWL or RDF Schema but it also deals with its evo-

lution (e.g. creations of a new property or extensions of an existing property hierarchy).

This evolution is handled by a set of rules enabling to adapt the database schema given modification patterns on the ontology schema. These patterns concern update operations at the terminological level of an ontology and can thus can only impact its sets of concepts and properties. Since `roStore` does not map ontology concepts to database relations, the database schema is not modified by insertions or deletions of concepts. For ontology properties (either datatype or object), the database schema needs to be updated. In order to present the different possible situations, we need to distinguish between different types of properties : a **stand-alone property** corresponds to a top property with no sub properties, a **top property** is a top property with sub properties and a **subsumed property** denotes a property having a super property which is already mapped to a database relation and no sub properties.

The kind of update operations we are interested in are limited to insertion, deletion and renaming of ontology properties. That is, we do not consider updates of a property's domain or range since it does not impact the table organization of the relational database. Moreover, we do not describe the semantics of modification operations (e.g. modification of the top property of a given hierarchy with an already existing property) since it can be specified in terms of the semantics of insertion and deletion operations. Nevertheless, for optimization reasons, this is not the operational semantics implemented in `roStore`.

We now detail the operations associated to evolution scenarios :

1. insertion of a stand-alone property P_1 : a new relation named P_1 is created in the database schema and this relation has only `subject` and `object` attributes.
2. insertion of subsumed property P_2 (with P_1 has a super property) : subjects and objects involved in triples containing P_2 as a property will be inserted in the existing P_1 relation with P_2 stored in the `property` attribute. Note that if P_1 was a stand-alone property, a `property` attribute needs to be inserted in this relation and P_1 is the value associated to all tuples before insertions of triples involving P_2 otherwise this attribute was already present.
3. insertion of a top property P_3 on top of an existing property hierarchy (with former top property P_1) : the relation P_1 is renamed after P_3.
4. insertion of a top property P_4 which is the top property of a hierarchy

together with another property P_3. This corresponds to multiple inheritance case and can be handle as follows : it is the end-user's decision to rename P_3 after P_4 or leave the schema unmodified.

5. removal of a stand-alone property P_5 : the relation corresponding to this property is deleted.
6. removal of a subsumed property P_6 : all tuples in relation P with `property` attribute equal to P_6 are deleted from the database.
7. removal of a top property P_7 : if P_7 has a single direct descendant, then relation P_7 is renamed after it. Otherwise, the end-user selects one of the descendants to rename relation P_7.
8. renaming of a stand-alone property P_8 into P_9 : the database relation P_8 is renamed into P_9.
9. renaming of a top property P_{10} onto P_{11} : the database relation P_{10} is renamed into P_{11} and all labels P_{10} in the `property` attribute of the relation are modified to the label P_{11}.
10. renaming of a subsumed property P_{12} into P_{13} : no database relation renaming is necessary, only the transformation of the label P_{12} into P_{13} in the `property` attribute of the relation is necessary.

4. Inference-based data management

Agents, i.e. applications or end-users, provide the `roStore` system with SPARQL queries (Prud'hommeaux & Seaborn 2008). The system then goes through a translation step to generate automatically a set of SQL queries which are going to be executed on the underlying RDBMS. A large body of work has already been done on the SPARQL to SQL translation topic but our contributions are tailored toward the specificities of both the vertically partitioned (VP) and `roStore` layouts. Moreover, it supports several semantic query rewriting techniques to enable TBox inferences.

In a nutshell, SPARQL is a graph-matching query language for RDF which has been released by the W3C. A query is composed of a pattern-matching part taking the form of triples. Natively, SPARQL does not support any form of inference hence it must rely on inferences performed by an external reasoner, e.g. Pellet. Most frequently, it results in generating a set of SPARQL queries integrating the inferred results. In the section, we present an approach based on this principle and which is extended with features for (i) ensuring a

declarative approach of specifying and processing queries and (ii) optimizing the set of generated SQL queries.

4.1. SPARQL extensions and translation to SQL

In order to propose a declarative approach for defining SPARQL queries, we propose to extend this query language with several operators. These operators can be attached to any of the three positions of a triple. Intuitively, they specify the inference mechanisms which can be associated to concepts and properties found in SPARQL triple patterns. These inferences are related to concept and property hierarchies and enforce the generation of SQL queries. These operators correspond to :

- `@subConcept` and `@superConcept` which enforce to compute the sub(respectively super) concept hierarchy of a given concept.
- `@subProperty` and `@superProperty` which enforce to compute the sub (respectively super) property hierarchy of a given concept.

Of course, these operators can be combined over a single SPARQL query and chained together, e.g. `worksFor@subProperty@superProperty` to retrieve all the properties of the property hierarchy containing `worksFor` (evidently this query extract is equivalent to `memberOf@subProperty`). The following example presents a concrete example over the LUBM ontology.

Example 2 : Suppose we would like to retrieve all subjects involved in triples with a property of the `memberOf` property hierarchy and whose (RDF) type is any of the concepts of the `Professor` concept hierarchy. This SPARQL query takes the following form :

```
SELECT ?s WHERE { ?s memberOf@subProperty ?o.
?s rdf:type Professor@subConcept.}
```

which enables to generate the following SQL query :

```
SELECT subject FROM memberOf m JOIN type t
ON (t.subject=m.subject) WHERE t.object
IN ('Professor','FullProfessor',..);
```

4.2. Domain and range based inferences

The SPARQL extensions we have presented involve Description Logics (DL) standard inferences, namely concept subsumption, to enhance data retrieval and update. In this section, we present another form of reasoning based on the types associated to the domain and range of a given property. Intuitively, the main idea is (i) to test the satisfiability of a given SPARQL query

by checking domains and ranges of involved properties and (ii) to optimize the set of generated SQL queries by generating only satisfiable queries.

As a consequence, these mechanisms enable to detect unsatisfiable queries (i.e. queries that return empty result sets independently of the database instance) and to generate only satisfiable SQL queries when inferences on concept and property hierarchies are involved. These mechanisms can have a big impact on the query execution performance since it can avoid to perform a complete scan over a set of property tables. To this end, this approach can be compared to indexing techniques found in RDBMS. But our solution is dynamic rather than static and does not require indexes to be persistent, hence opening perspectives for better update query performances.

5. Evaluation

5.1. Experimental settings

The experiments we are presenting in this section have been conducted on four synthetic databases generated using LUBM. The RDF data sets have been translated into different physical organization models. They are decomposed into the two approaches `vpStore` and `roStore`. In order to emphasize the efficiency of our solution on queries needing reasoning, we had to test these settings in a context similar to (Abadi & Marcus 2007) and also to the setting adopted in (Curé & Faye 2010) for evaluation of SELECT queries. More precisely, we evaluated each approach on a row-oriented and a column-oriented RDBMS. This yields the four following approaches : `vpStore` resp. on a row (**vpRow**) and column (**vpColumn**) stores and `roStore` resp. on a row (**roRow**) and column (**roColumn**) stores. Hence a total of sixteen databases have been 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 (Abadi & Marcus 2007)) essentially due to (i) the lack of maintenance of the latter one, (ii) the open-source licence of MonetDB and (iii) 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 Pentium 4 (2.8 GHz) operated by a Linux Ubuntu 9.10, with 1 Gbytes of memory, 1MB L2 cache and one disk of 60 Gbyte spinning at 7200rpm.

The disk can read cold data at a rate of approximatively 55MB/sec.

For the `vpRow`, there is a clustered B+ tree index on the *subject* and an unclustered B+ tree on the *object*. Similarly, for the `roRow`, a clustered B+ tree index is created on the *property* column and unclustered B+ trees on the *subject* and *object*. As noted in (Sidirourgos & Goncalves 2008), MonetDB does not include user defined indices. Hence, we relied on the ordering of the data on *property*, *subject* and *object* values. More precisely, any two columns table of `roColumn` and `vpColumn` is ordered on *subject* and *object* ; while any three columns table (of `roColumn`) is ordered on *property*, *subject* and *object*. This evaluation does not consider structures common to `vpStore` and `roStore`, i.e. relations composed of only two attributes. Hence all queries executed in this evaluation section consider relations generated for property hierarchies.

In this evaluation we are only interested in how update queries perform considering `vpStore` and `roStore`. We consider that this particularly relevant at the time when SPARQL 1.1, which integrates update facilities, is at the W3C Working Draft stage. To this end, we have defined a set of 7 queries considering all update forms (i.e. delete, update and insert). The first three queries (Q1 to Q3) operate respectively deletion, insertion and modification of a large number of tuples on each database while queries Q4 to Q7 operate on smaller set of tuples. All queries involve the `department0` instance over our four synthetic databases. Volumes are presented in the following table :

Universities	'Department0' instances	Total number of triples
1	720	100868
2	1158	236336
5	2814	643435
10	5633	1296940

Q1 : delete all instances of `Department0` from the `memberOf` properties (i.e. `headOf, worksFor and memberOf` properties.

Q2 : restores all instances of `Department0` of the `memberOf` properties. This is the same pattern as in the previous query.

Q3 : modifies all instances of `DepartmentO` of a given university to `Department0` to another university.

Q4 : This is an update query replacing `FullProfessor9` by `FullProfessor4` as the new head of `Department0` at `University0`.

Q5 : inserts a new professor as the head of `Department0` of `University0`.

Q6 : All students of `Department0 University0` are now members of `Department1 University0`. This query requires inferences since all sub classes of `Student` are considered.

Q7 : All persons which are member of `Department0` at `University0` are 'moving' to `Department0` at another University.

5.2. Experimental results

Q1 : The `roStore` approach is faster than `vpStore` since only one query is executed on the former and three for the latter.

Q2 : Both `roStore` approaches are faster than the `vpStore` solutions. Again, this is due to the number of queries executed.

Q3 : `roStore` is faster than `vpStore` on both column and row-oriented databases. This is again related to the number of queries executed for each approach (3 for `vpStore` and 1 for `roStore`). There is not much difference between the `vpRow` and `vpColumn` since the condition of the query is performed on an attribute which is not used in an index. Remember that the implementation of the column-oriented database we are using does not support indexes but relies on ordering of data within a table. This may impair performances after a large number of updates.

Q4 : On the one hand, the `vpStore` approach requires two queries since a DELETE operation is required on the `worksFor` relation and an INSERT operation is needed on the `headOf` relation. On the second hand, the `roStore` approach needs a single UPDATE query since on the `memberOf` relation. This justifies the better performances of `roStore` over `vpStore`. This query also emphasizes the impact of indexed attributes. For the `vpStore` databases, the row version exploits an index and is much faster than its column counterpart. The UPDATE query of the `roStore` approach selects precisely a given tuple with a selection over the three attributes of the `memberOf` relation. Since this relation is ordered on the column version, the differences are not that important between the column and row approaches.

Q5 : the `roStore` and `vpStore` approaches both contain a single query. The `vpStore` query accesses the `headOf` relation which contains a subset of the `memberOf` relation accessed by the `roStore` approach. But to analyze the query results, it is also important to consider the index maintenance of row stores which is not necessary for column stores at the cost of degrading performances since the relation will not be sorted after many updates.

Q6 : Both `vpStore` and `roStore` need a single UPDATE query with in-

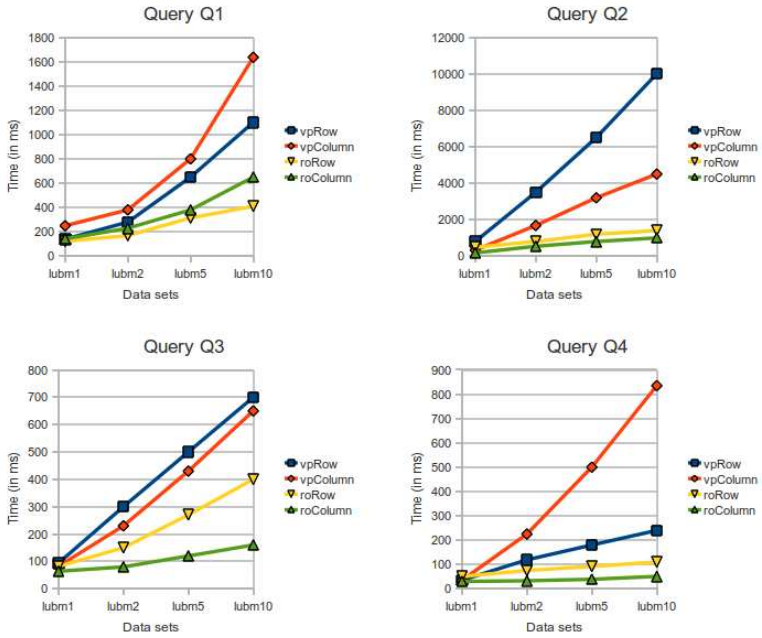

FIGURE 2: Queries involving large volumes of data

ferences (retrieving all sub concepts of the `Student` class). Unsurprisingly, all query performances are very close with a slight advantage of `vStore` other `roStore` (since no selection is required on the property attribute for `vpStore`)

Q7 : The `vpStore` version requires 3 queries (one for each property of `memberOf` property hierarchy) while the `roStore` version requires a single query since it only accesses the `memberOf` relation. The same form of inferences is required on all queries, i.e. inferring all sub concepts of the `Person` concept.

6. Conclusion

As a logical data model, RDF does not propose a preferred physical storage approach. Depending on the queries asked by applications and the structure of the underlying ontology, several data models and physical organizations are possible. Our `roStore` system, originally presented in (Curé & Faye 2010), corresponds to an ontology-guided solution which exploits property hierarchies to partition the RDF triples into several relations. This approach

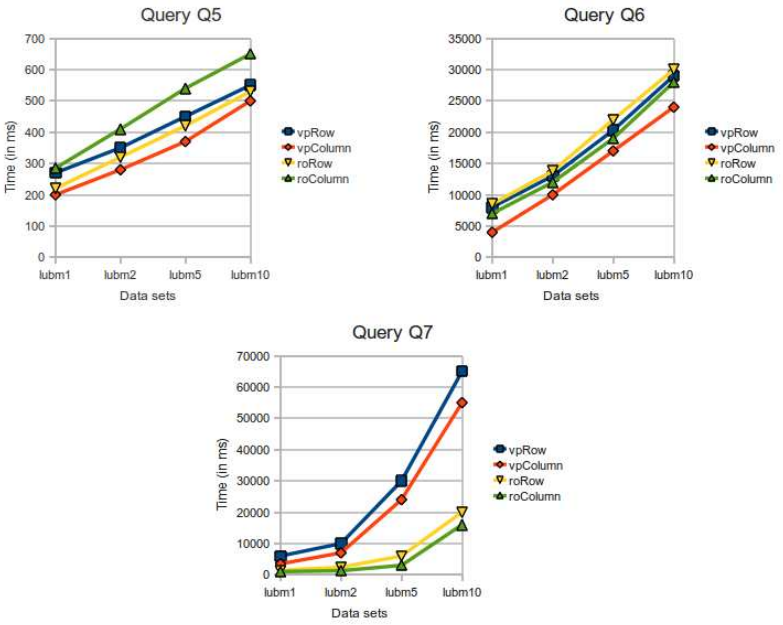

FIGURE 3: Queries involving few tuples

has been shown to be particularly efficient when implemented on top of a column oriented RDBMS (as opposed to row oriented one). An obvious question related to the characteristics of column oriented RDBMS is : how is it performing on update queries ? The evaluation we have conducted emphasizes that performances are comparable to its row oriented counterpart and that `roStore` stands as a good competitor to vertical partitioning. We now have a clear picture on two solutions of ontology-guided partitioning of RDF triples : whenever your ontologies contain property hierarchies and your applications make an extensive use of them, the `roStore` approach may be advantageous even when update operations are frequently executed. Moreover, in this paper, we have extended our previous work on `roStore` by (i) proposing an extension to the SPARQL query language that enables an inference-based generation of SQL queries and (ii) providing a set of rules to modify automatically the relational schema when the ontology is modified. We consider that together with (Curé & Faye 2010), this paper proposes a clear overview of `roStore` with evaluations of both retrieving and updating queries, a schema evolution approach and a semantic query rewriting solution.

In future works, we aim to tackle so called NoSQL databases to store RDF

triples. We are confident that hybrid solutions between standard RDBMS and NoSQL databases will enable to design scalable Semantic Web applications.

7. References

Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K. : Scalable semantic web data management using vertical partitioning. VLDB '07, 411-422, 2007

Curé, O., Faye D. , Blin, G. : Towards a better insight of RDF triples Ontology-guided Storage system abilities. 6th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS'10). Shanghai, China. 2010.

Guo Y., Pan Z., Heflin J. : LUBM : A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3) :158–182, 2005

Harris, S., Gibbins, N. : 3Store Efficient bulk RDF storage. PSSS'03, 1–20, 2003

Harth, A., Decker, S. : Optimized Index Structures for Querying RDF from the Web. LA-WEB '05, 71–80, 2005

Kolas, D., Emmons, I., Dean, M. : Efficient Linked-list RDF Indexing in Parliament. SSWS'09, 17-32, 2009

Erling, O., Mikhailov, I. : RDF Support in the Virtuoso DBMS. Conference on Social Semantic Web 2007. 59-68

McGlothlin, J., Khan, L. : RDFJoin : A Scalable of Data Model for Persistence and Efficient Querying of RDF Datasets. Technical Report UTDCS-08-09. http ://www.utdallas.edu/ jpm083000/rdfjoin.pdf.

Neumann, T., Weikum, G. : RDF-3X : a RISC-style engine for RDF. VLDB Endow. 2008. 647-659.

Prud'hommeaux, E., Seaborn A. : SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/` 2008

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

Weiss, C., Karras, P., Bernstein, A. : Hexastore : sextuple indexing for semantic web data management. VLDB'08, 1008-1019, 2008