Towards a better insight of RDF triples Ontology-guided Storage system abilities

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, considering reasoning dependent queries, the problem of providing an efficient, in terms of performance, scalability and data redundancy, partitioning of the data is still open. In regards to recent data partitioning studies, it seems reasonable to think that data partitioning should be guided considering several directions (e.g. ontology, data, application queries). This paper proposes several contributions: describe an overview of what a roadmap for data partitioning for RDF data efficient and persistent storage should contain, present some preliminary results and analysis on the particular case of ontology-guided (property hierarchy) partitioning and finally introduce a set of semantic query rewriting rules to support querying RDF data needing OWL inferences.

1 Introduction

The generally encountered use of ontologies consists in performing data inferences and validation using a Semantic Web compliant reasoner. The corresponding reasoning mechanism can be used to generate a set of queries executed over the appropriate data sets. For example, this approach was designed in a medical application [8] where inferences on chemical molecules were needed to highlight contra indications, side effects of pharmaceutical products. As mentioned in [8], results of queries with both inference on property (*i.e.* `rdf:property`) and concept (*i.e.* `rdf:class`) hierarchies are required by the application as well as by data quality or data exchange external tools.

In regards to large ontologies (e.g. OpenGalen or SNOMED in the medical domain) and data sets (e.g. Linked Data), providing efficient performances to reasoning dependent queries is an important issue. We believe that to enable efficient response time to such queries, one has to give a special attention to the storage system associated to the triples. In fact, RDF is basically a data model and its recommendation does not guide to a preferred storage solution. 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. In this paper, we do not focus on the latter one.

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 based on partitioning (cf. Figure 1).

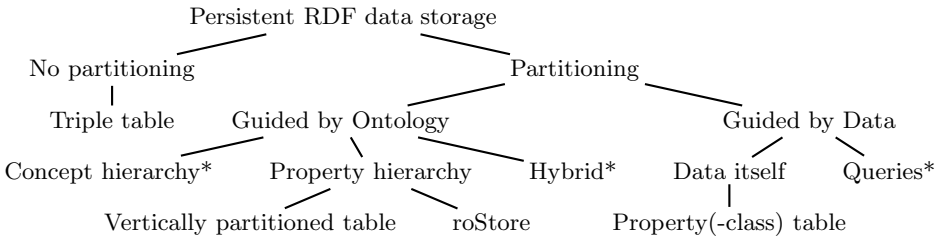


Fig. 1. Physical organization of RDF data based on partitioning. (*) no known study yet.

On one hand, there exists tools such as Sesame [5], Jena [18], Oracle [6] and 3store [10] which rely on a straightforward mapping of RDF into an RDBMS – called *triple table* approach. Each RDF statement of the form (*subject*, *predicate*, *object*) is stored as an entry of one large table with a three-columns schema (*i.e.* a column for each the *subject*, *predicate* and *object*). Indexes are then added for each of the columns in order to make joins less expensive. However, since the collection of triples are stored in one single table, the queries may be very slow to execute. Indeed when the number of triples scales, the table may exceed memory size (inducing costly disk-RAM transfers). Nevertheless, simple *statement-based* queries can be satisfactorily processed by such systems, although they do not represent the most important way of querying RDF data. Still, this storage system scales poorly since complex queries with multiple triple patterns require many self-joins over this single large table as pointed out in [18, 16, 12].

Whereas this specific approach does not use partitioning at all, on the other hand, some recent research highlighted two efficient main trends depending on the information one uses to guide the partitioning: guided by (1) the underlying ontology or (2) the data itself. Intuitively, one would expect that a well suited data partitioning will induce a better response time to queries (at least SELECT ones). Indeed, data partitioning will allow queries to be made on smaller sets of entries which, given an adapted RDF data clustering, should be faster. The counterpart of this storage system will be some possible worst performance for data updates.

Considering partitioning guided by the underlying ontology, most of the recent works focus only on the property hierarchy. Among others, the *vertical partitioning* approach suggested by Abadi et al. in [2] is to be mentioned. In this approach, using a fully decomposed storage model (DSM) [7], the RDF data is divided into n two columns (*subject,object*) tables where n is the number of unique *predicates* in the data. Each of these resulting tables represents a particular *predicate*, with an entry for each statement of the data containing the corresponding *predicate*. Sorting the tables according to the *subject* allows fast merge joins for reconstructing information about multiple *predicates* for subsets of *subjects*. The vertically partitioned approach offers a support for multi-valued attributes. Indeed, if a *subject* has more than one *object* for a given *predicate*, each distinct value is listed in a successive row in the corresponding table. For a given query, only the *predicates* involved in that query need to be read. Finally, the building of the two-columns tables can be done easily – without a clustering algorithm – by only browsing and relying on the property hierarchy of the data. As previously mentioned, as a counterpart, data updates/insertions may be slower in vertically partitioned tables rather than triple ones since multiple tables need to be accessed for statement about the same *subject*.

In [2], the authors described how a column-oriented DBMS [15] (*i.e.*, a DBMS designed especially for the vertically partitioned case, as opposed to a row-oriented DBMS, gaining benefits of compressibility [3] and performance [1]) can be extended to implement the vertically partitioned approach. Roughly, this is done by storing tables as collections of columns rather than collections of rows. The goal is to avoid transferring entire rows 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. Note that, in an independent evaluation [14] of the techniques presented in [2], the authors pointed out potential scalability problems for the vertically partitioned approach when the number of *predicates* in an RDF data set is high. With a larger number of *predicates*, the triple table solution manages to outperform the vertically partitioned one.

Depending on the type of reasoning dependent queries, it may be more efficient to consider an intermediate model between triple tables and vertical partitioning approaches. A first contribution of this work will be to provide an analysis of the effectiveness of an intermediate approach – also based on property hierarchy – where a table is created only for *top predicates*. Another interesting track (for future investigation), would be to consider partitioning the data regarding the concept hierarchy rather than the property one and/or considering both. To the best of our knowledge, such study has not yet been conducted. Considering data guided partitioning, one may distinguish two types of guides. First, one may consider that the RDF data itself may induce an efficient partitioning. The main achievement for this type of guide is the *property table* technique which was introduced later on [17] 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 [17, 6]. Indeed, each named table includes a *subject* and several fixed *predicates*. The main idea is to discover clusters of *subjects* often appearing with the same set of *predicates*.

A variant of the property table named *property-class table* uses the `rdf: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 *predicates* will be defined for all *subjects*. 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`, it is common to seek for all defined *predicates* of a given *subject*, which, in the property table approach, requires scanning all tables.

Note that, in this approach, adding *predicates* 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. Another type of guide which may worth being studied is *queries*. Indeed, one may consider an evolutive storage schema in regards to recent queries made on the data. To the best of our knowledge, this track of research has also not been considered yet.

In this article, we will concentrate on giving some preliminary results (on medium-sized datasets) on an intermediate property hierarchy based approach (that will need to be pursued) – namely RDF Ontology-guided Storage system (`roStore`). After presenting the general approach (Section 2), we will, in Section 3, evaluate the efficiency difference with vertically partitioning on the LUBM benchmark over both row and column oriented databases and on some extra specific queries highlighting limits of vertically partitioning.

2 The `roStore` approach

As a first step to an efficient `RDF` storage road map, we propose an intermediate ontology-guided approach – namely `roStore` – which lies between the two extremes: triple and vertically partitioned tables. The aim of this approach is to try to analyse the efficiency of a compromise approach where less partitions are used. Intuitively, such physical organization will take benefits of requiring less joins in practical queries and with less risk of unmappable table in memory.

As already mentioned, we believe that there should not be a unique generic solution to `RDF` storage and that depending on the data itself, the underlying ontology, application queries, better performance may be obtained by considering alternative and several dedicated approaches. The major aim of `roStore` is to

provide some clue of this belief. We will demonstrate that `roStore` is one of them and may, in specific cases, induce better efficiency. In this context, we consider `roStore` as **one among other** interesting physical organizations based on property hierarchy that should be present in the `RDF` storage road map.

Our storage approach derives from the vertically partitioned one and extends this last by putting back together into a single table data related to a *top-property* of a property hierarchy. Given a hierarchy, we say that a *predicate* is a top-property if it is only an `rdf:subPropertyOf` of itself. For each such 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 (cf. Figure 2).

Let us first notice that, providing with this definition, any *predicate* that is not an `rdf:subPropertyOf` of a top-property will still be stored in a two-columns table. This induces an insignificant expense of the space complexity of this novel approach. Moreover, in case of a cyclic *property* hierarchy, all *predicates* are necessarily all semantically equivalent. Hence selecting a single canonical *predicate* and rewriting triples accordingly is sufficient. Despite the fact that considering top-property seems to be the most natural, one may, depending on the topology of the hierarchy, define other physical organizations inducing better performance too for specific cases. Our preliminiary results demonstrate that an evolutive physical organizations guided by the queries may be efficient. The main impact of merging some tables is obtaining 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 information concerning a family of *predicates* of the property hierarchy; since they will be quite related. In the following, we will denote by `vpStore` (resp. `roStore`) the vertically partitioned (resp. our) approach.

Example 1: Let us consider a small data set (Figure 2b) defined over a given property hierarchy (Figure 2a). With `vpStore`, the triples would be distributed over six different tables as displayed in Figure 2c. Comparatively, in `roStore`, one obtains only two different tables (Figure 2d): a single relation named after the top-property `pa` and a relation named after the property `pf`.

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. `pd` with `vpStore` in Example 1). Moreover, the set of tuples stored is the same as 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 `pa` hierarchy. Considering `vpStore`'s physical design, the following `SQL` query is needed:

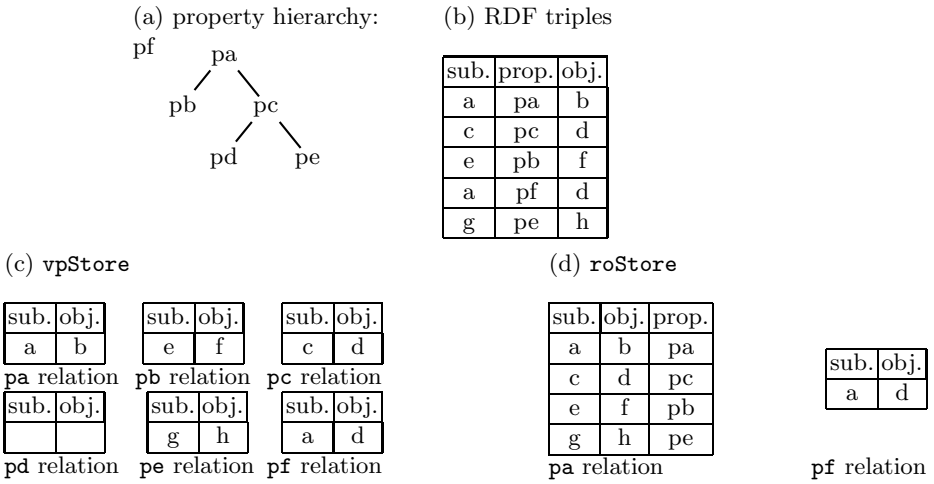


Fig. 2. Storage comparison of `vpStore` and `roStore`

```
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 considering `roStore`'s physical design with:

```
SELECT object FROM pa;
```

Such example highlights the kind of (1) reasoning dependent queries and (2) corresponding improvement one can obtain by using an intermediate physical organization such as `roStore` over `vpStore` when property hierarchies are present in the ontology.

In order to analyse more deeply the corresponding efficiency of `roStore` approach, we will first compare it to the `vpStore` approach on the LUBM benchmark and on some specific queries that highlight limits of vertically partitioning. In this work, as a first contribution, we focus only on SELECT queries. We are currently investigating the possibly negative impact of partitioning on UPDATE queries. As far as we went on this track, it seems that this impact is reasonable. First, we will discuss how to take benefits of the ontology based structure of the data without the needs of heavy inference mechanisms.

Indeed, compared to classical table schema, the ontology is far more meaningfull and can thus be used to enhance the performance even without needing knowledge inference. We propose an efficient use of Semantic Query Rewriting (SQR) adaptable to and usable by most of the data storage approaches. Our semantic query rewriting aims are, first, to guarantee the exhaustiveness of results returned when requiring data that should include `rdf:subClassOf` and `rdf:subPropertyOf`; and, a query validation mechanism simply based on the

domain and range information related to *predicates*, *i.e.* resp. `rdfs:domain` and `rdfs:range`. One has to note that the mechanisms we propose are not really needing heavy reasoning nor data inference mechanisms. Indeed, they can be considered as an efficient use of the right-away available information of the ontology.

The semantic aspect of this rewriting is provided by a thorough usage of the `OWL` entailment mechanism, on one hand, to detect if the answer set of a query will be empty or not, on the other hand, to optimize query in order to guarantee exhaustiveness of the solution returned. The rules can be decomposed into two sets: (i) a set of rules, denoted `subsume`, dealing with concept and property subsumptions; (ii) a set of rules, denoted `propertyCheck`, dealing with the `rdfs:range` and `rdfs:domain` of a given *predicate*. The rules processed by the `subsume` procedure are using the `OWL` inferences to compute all the sub-concepts (resp. sub-properties) of a given concept (resp. property). In fact, the query studied in Example 1 was already using the `subsume` procedure.

Example 2: Consider that the `rdfs:range` of the *predicate* `pb` of Example 1 is of `rdf:type ClassA` which is the top-concept in the following concept hierarchy:

$$\texttt{ClassC} \sqsubseteq \texttt{ClassA},\ \texttt{ClassB} \sqsubseteq \texttt{ClassA}\ \textit{and}\ \texttt{ClassC} \sqsubseteq \neg\texttt{ClassB}$$

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 *predicate* and all *subjects* belong to the `ClassA` hierarchy. Using `subsume`, the query can be translated in the following `SQL` query:

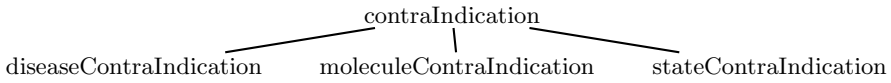
```
SELECT subject, object FROM pa, type WHERE type.subject =
pa.subject AND pa.property = 'pb' AND type.object IN
('ClassA','ClassB', 'ClassC');
```

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

The rules of `propertyCheck` are being processed as follows: first the `SPARQL` query is parsed and for each *predicate* explicitly mentioned in the query with a typed (`rdf:type`) *subject* or *object*, we store a structure containing the *predicate* name and the `rdf:type` 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 `rdfs:domain` (resp. `rdfs:range`) defined in the ontology for this property.

Example 3: Let us consider the property hierarchy of Figure 3, dealing with contra indications and the corresponding `roStore` organization.

Moreover, consider the following ontology axioms: (1) `rdf:range` of `disease` `ContraIndication` is an instance of the `Disease` concept, (2) $\texttt{Disease} \sqsubseteq \texttt{Top}$, (3) $\texttt{Molecule} \sqsubseteq \texttt{Top}$ and (4) $\texttt{Disease} \sqsubseteq \neg\ \texttt{Molecule}$. Intuitively, axioms (2) to (4) state that the `Disease` and `Molecule` concepts are a sub-concept ot the `Top` concept and are disjoint. Consider the following `SPARQL` query:



subject	object	property
Ibuprofen	Ticlopidin	moleculeContraIndication
Ibuprofen	Clopidrogel	moleculeContraIndication
Ibuprofen	Breast feeding	stateContraIndication
Ibuprofen	Pregnant	stateContraIndication
Ibuprofen	Hypertensive heart	diseaseContraIndication

Fig. 3. Sample of the contraIndication relation

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

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

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

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

The query is satisfiable since there is a 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. Note that this simplification does not work for property with multiple-range/domain. Those examples demonstrate that it is worth to efficiently use the basic knowledge available directly in the concept and property hierarchies.

3 Evaluation

3.1 Experimental settings

All our experiments have been conducted on four synthetic databases. They all have been generated from the Lehigh University Benchmark (LUBM) [9] 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 (*i.e.* relating objects to objects) and 7 data type properties (relating objects to literals).

This ontology serves as the schema underlying the four 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 data sets

DB name	# Universities	# Concept instances	# Property instances	# Triples
lubm1	1	15195	60859	100868
lubm2	2	62848	189553	236336
lubm5	5	114535	456137	643435
lubm10	10	263427	1052895	1296940

The `RDF` data sets are later translated into the different physical organization models we would like to evaluate. They are decomposed into the two main 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 [2]. More precisely, we evaluated each approach on a row store and a column store RDBMS. This yields the four following approaches: `vpStore` resp. on a row (**vpRStore**) and column (**vpCStore**) store and `roStore` resp. on a row (**roRStore**) and column (**roCStore**) store. Hence a total of sixteen 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 [2]) 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 Pentium 4 (2.8 GHz) operated by a Linux Ubuntu 9.10, with 512 Mbytes 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 `vpRStore`, there is a clustered B+ tree index on the *subject* and an unclustered B+ tree on the *object*. Similarly, for the `roRStore`, a clustered B+ tree index is created on the *property* column and unclustered B+ trees on the *subject* and *object*. As noted in [14], 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 `roCStore` and `vpCStore` is ordered on *subject* and *object*; while any three columns table (of `roCStore`) is ordered on *property*, *subject* and *object*.

Our evaluation contains fifteen queries out of which eleven are coming from the LUBM benchmark and four tackling the LUBM ontology to evaluate some particular aspects of `roStore`. An interesting aspect using LUBM Benchmark queries is that do not aim to emphasize on the performances of a given storage

model. Moreover, these queries tackle a wide range of possibilities on volume of input (number of tuples retrieved) and selectivity rate (*i.e.* number of conditions in the `WHERE` clause of a query). Among the eleven evaluated queries, three do not require any form of reasoning (#1, #2 and #14) and the eight remaining queries can be divided in two groups whether they are involving reasoning on the concept hierarchy (#3,#4,#6,#7,#9,#10) or both concept/property hierarchies (#5,#8). We now present the purpose of each of these queries:

Q1: retrieves instances of the `GraduateStudent` class who have taken the course `http://www.Department0.University0.edu/GraduateCourse0`.

Q2: retrieves three instances of respectively the `GraduateStudent`, `University` and `Department` concepts for those students that are member of a department, this department is a sub-organization of a University and this student has an undergraduate degree from this university.

Q3: selects all kinds of publications which have been authored by a given assistant professor.

Q4: retrieves all kinds of professors, their name, email address and telephone number for those professors working for a given department.

Q5: the result contains instances of the `Person` concept hierarchy for those persons that are related to a given department by either the `memberOf`, `workingFor` or `headOf` properties.

Q6: displays URIs of instances of the `Student` concept hierarchy.

Q7: retrieves instances of all kind of students and all kinds of courses for courses that are related by the `takesCourse` property for those courses that are taught by a given professor.

Q8: displays instances of all kinds of students with their email addresses and department instances of a given university these students are member of.

Q9: the retrieved dataset contains instances of the `Student`, `Faculty` and `Course` concept hierarchies for those students that are advised by faculties, have taken some courses taught by those faculties.

Q10: selects instances of all the `Student` class hierarchy who have taken a given course.

Q14: selects all undergraduate students.

We have introduced **Q15** to emphasize `roStore` performances when values are needed for a property hierarchy. In fact, it retrieves all *subjects* involved in triples where the *predicate* is one of the properties of the `memberOf` property hierarchy, i.e. `memberOf`, `headOf` and `worksFor`. This query is similar to Q5 but does not refer to any concepts.

Finally, the following three queries aim to highlight the efficiency of our SQR approach. **Q16:** selects the *subject* and *object* in triples where the *predicate* is `teacherOf` and *subject* is of `rdf:type` AdministrativeStaff. This query returns an empty answer set since the `rdfs:domain` of `teacherOf` is the `Faculty` concept which is disjoint with `AdministrativeStaff`. In the next section, we confront the performances of this query to the simple detection of unsatisfiability of our SQR solution.

Q17: selects the *subject* and the *object* in triples where the *predicate* is `teacherOf` and *subject* is of `rdf:type Faculty`. This query requires a join.

Q18: has the same purpose as Q17 but exploits one of our rewriting rules to improve its performances. In fact, the join in Q17 is not necessary if one knows that the `rdfs:domain` of `teacherOf` is the concept `Faculty`.

In the experiments, we will store the LUBM ontology in main-memory and perform reasoning using the Jena framework. We provide more details concerning the experimental settings and results on the following web site:
`http://sites.google.com/site/wwwrostore`.

3.2 Experimental results

The results presented in this section correspond to the average of 5 hot runs (i.e. repeated runs of the same query without stopping the DBMS) of real time (i.e. execution time of a query defined as the wall clock passed between the server receiving the query and before returning the results to the client) executions. All performance times, except for query Q16 and Q17, include the time needed to perform the inferences. Finally, in order to highlight the differences in terms of performances between the various approaches, we either present the results in bar or line diagrams.

Analysis of Q1. Not surprisingly, column stores outperform row stores. Indeed, the results will only contain a unique column which will clearly benefit column store advantage. Moreover, since the query does not involve sub-properties, the performances of `vpStore` and `roStore` are quite similar.

Analysis of Q2. This time, the row stores are more efficient than the column ones. The results require, in this case, to retrieve two columns of three tables, hence in a row store both columns will be transfered from the hard drive to main memory in a single step while two transfers will be needed for column stores. Moreover, two out of these three tables corresponds to *predicates* being part of a group of related predicates in the property hierarchies, namely `memberOf` and `undergraduateDegreeFrom`. Since we voluntarily decided to perform no inferences on these two *predicates*[3] (*i.e.* not including sub-properties in the query), it is not surprising that `vpStore` outperforms `roStore` since each *predicate* corresponds in the `vpStore` to a table.

Analysis of Q3. Despite the fact that this query has a similar structure as Q1 (*i.e.* only two triples are present in the `WHERE` clause), it requires to retrieve all concepts of a wide hierarchy. Due to the ordered organization of tuples, column stores outperform row ones which rely on indices and on a less effective I/O transfers. In a similar manner to Q1, the difference between `vpStore` and `roStore` is not significant.

Analysis of Q4. Once again, in this query, we do not use inference on the `worksFor` *predicate* to include results on sub-properties of this last. This is motivated by the will to emphasize on the weaknesses of the `roStore` approach. As expected, `vpStore` is, in this context, outperforming `roStore`. In fact, even

[3] since it will be specifically considered in query Q15.

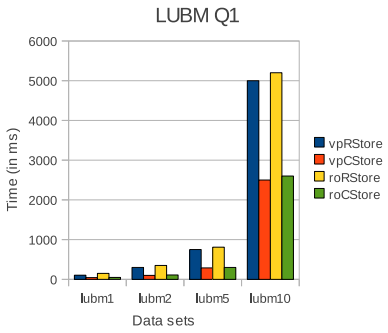


Fig. 4. Performance results for Q1

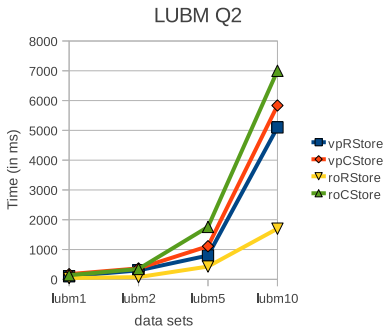


Fig. 5. Performance results for Q2

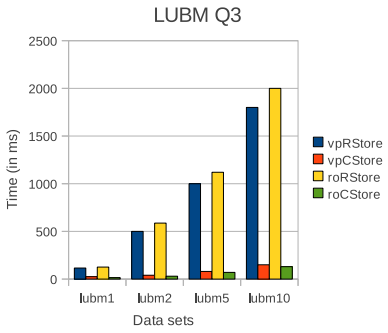


Fig. 6. Performance results for Q3

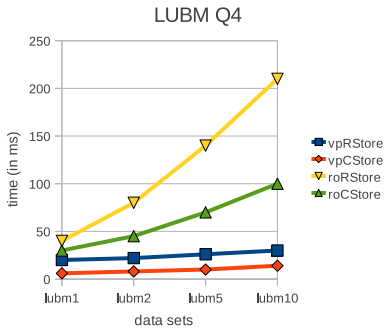


Fig. 7. Performance results for Q4

`vpRStore` is outperforming `roCStore`; which can be induced by the high selectivity nature of the query (four attributes in result set).

Analysis of Q5. Due to the exploitation of the sub-properties of the *predicate* `memberOf` in this query, it is not surprising that `roStore` outperforms `vpStore`. Indeed in `vpStore`, the results of the query comes from the union of three distinct queries (one for each *predicates* involved) while `roStore` only requires a single query.

Analysis of Q6. This query retrieves the subjects from a two columns table (i.e. type). Because the column stores primarily order these relations on the subject, they are more efficient than their row store counterparts. This is due to better I/O efficiency. Similarly to Q3, the `roStore` approach outperforms `vpStore`.

Analysis of Q7. Again query processes in `roStore` (resp. column store) is more efficient than `vpStore` (resp. row store). The reasons are similar to the ones for Q3.

Analysis of Q8. The analysis of the results for this query confirm the ones of Q5.

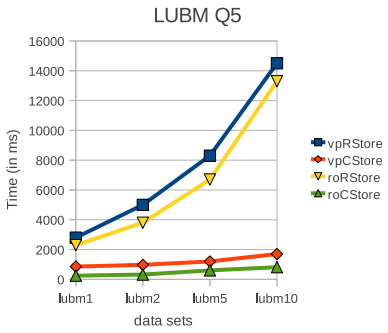


Fig. 8. Performance results for Q5

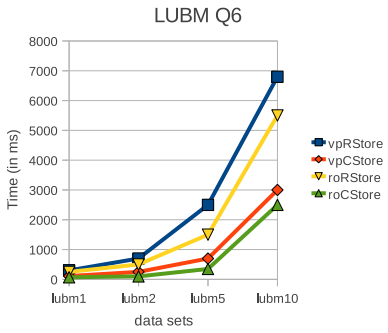


Fig. 9. Performance results for Q6

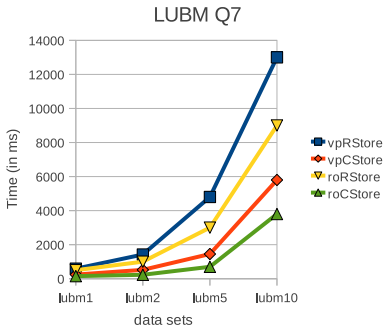


Fig. 10. Performance results for Q7

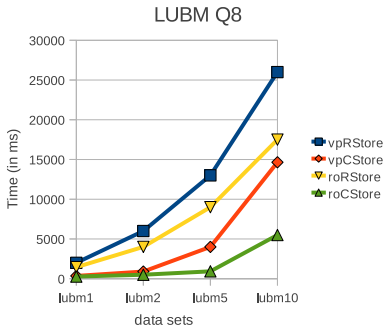


Fig. 11. Performance results of Q8

Analysis of Q9. This query does not require inferences on property hierarchies but some on several concept ones. As seen previously, in this situation column stores is more efficient than row stores. On column stores, `vpStore` and `roStore` have close performance results, with `roStore` slighty better than `vpStore`.

Analysis of Q10. The results are similar to Q9.

Analysis of Q14. This query has large input and low selectivity with no inferences. As expected, `roCStore` is faster than `vpCstore` which is more efficient than `roRStore`; the less effective being `vpRStore`. Note that this is due to distinguished variable being placed at the *subject* position of the only triple of the `WHERE` clause. A similar query pattern with the distinguished variable mapped to the *object* position of a triple would emphasize the superiority of the `vpStore` approach.

Analysis of Q15. This query clearly demonstrates the efficiency of `roStore` over `vpStore`. Even the row oriented `roStore` outperforms the column oriented `vpStore`. This is due to the presence of `UNION SQL` operators in the queries executed on the `vpStore` while `roStore` only requires a complete scan of the tuples of one table.

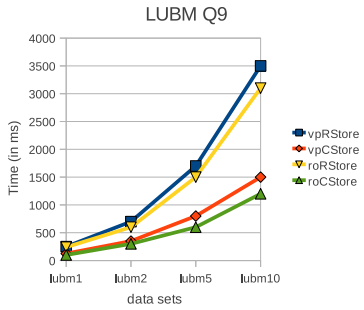


Fig. 12. Performance results for Q9

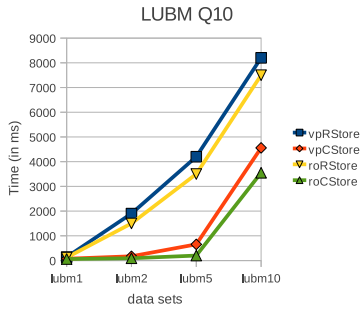


Fig. 13. Performance results of Q10

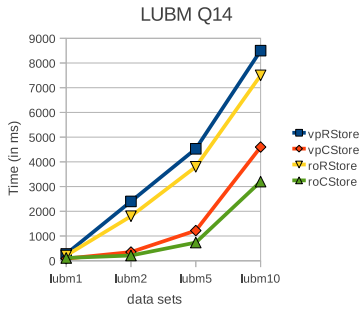


Fig. 14. Performance results for Q14

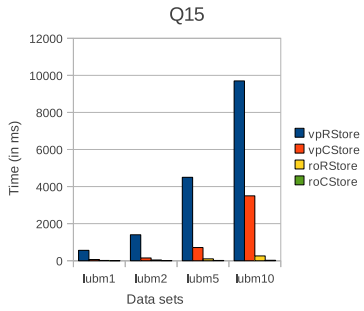


Fig. 15. Performance results of Q15

Analysis of Q16, Q17 and Q18. Finally, queries Q16, Q17 and Q18 emphasize the importance of reasoning over the ontology before executing queries over any of the store solutions. Figure 16 displays the duration times for all databases, ranging from approximately 42ms (column store with 1 university) to 1450ms (row store with 10 universities). This can be considered rather long to propose an empty answer set since, according to the ontology, the query is incoherent. Comparatively, the `propertyCheck` method we have implemented needs an average time of 1ms to reply that the query is coherent or not. Hence, a system implemented on top of an `OWL` compliant reasoner is able to determine almost instantly if the answer set is empty.

Moreover, it could also provide some explanations concerning the inconsistency of the query. We believe that such optimization are quite useful especially when end-users are not confident with all the details of a given ontology. The performance results of Q17 and Q18 are provided together in Figure 17 in order to highlight their comparisons. The purpose of Q17 and Q18 is to emphasize the importance of analyzing *predicate* `rdfs:domain` and `rdfs:range` in a property table approach. The execution of Q17 does not perform any optimization while Q18 checks that the concept `Faculty` is the `rdfs:domain` of the `teacherOf` *predicate* and hence a join to the `rdf:type` relation is not necessary.

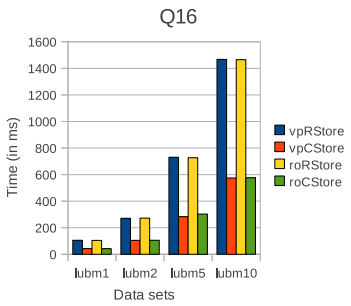


Fig. 16. Performance results for Q16

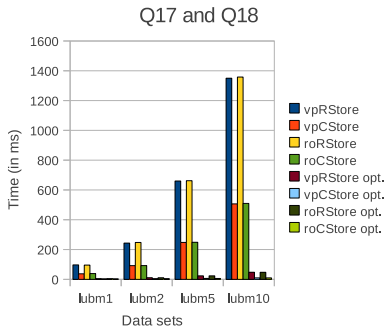


Fig. 17. Q17 and Q18 performance results

Summary: Several conclusions can be drawn from our evaluation. Considering the adoption of a database solution, we confirm the evaluations of [2] and [14] stating that column stores outperform row stores for `RDF` triple storage. The only exception in our experiments consists in Q2 which is rather due to the partitioning approach.

Concerning the partitioning approach, all our intuitions were confirmed by this evaluation. That is `roStore` outperforms `vpStore` whenever queries retrieve information from triples where properties belong a given property hierarchy (e.g. Q5 and Q15). On the contrary, `vpStore` is more efficient than `roStore` where only a subset of the properties of a property hierarchy are necessary to reply to a query (e.g. Q2). This result was expected since the `roStore` approach then requires to add additional conditions on the properties one wants to retrieve from a 'top property' relation.

Finally, the SQR approach seems to be quite useful since it does not slow down the processing of satisfiable queries and enables to detect unsatisfiable queries efficiently (e.g. Q17 and Q18). Anyhow, we consider that more evaluations need to be conducted on larger ontologies to confirm these results.

4 Conclusion

The first contribution of this paper is to show that depending on the type of applications and queries asked to the RDF triple stores, different partitioning approaches can be considered. Between the two extremes of triple and vertical partitioning, we introduced the `roStore` approach which is particularly advantageous for a certain class of queries, *i.e.* those relying on deep property hierarchies (e.g. the OpenGalen ontology contains a property hierarchy of depth 6). Moreover, this novel approach is a simple extension to the existing RDF column store work and can thus be easily adopted by other RDF stores. A second contribution of this work is to propose a semantic query rewritting solution that can be adopted by most of the RDF triples we have presented in this paper (triples tables, vertical partitioning, `roStore`, property-class tables). This approach seems

promising since it can be quite useful to detect unsatisfiable queries and optimizing other queries by analyzing property domains and ranges.

Our list of future works is large since we consider that several investigations need to be performed to complete the road map on efficient and persistent RDF triple storage. The first directions we would like to follow are ontology schema evolution in `roStore` (e.g. a new property hierarchy emerges or is removed from the ontology) and the consideration of concept hierarchies at the storage and querying levels.

References

1. Abadi, D.J., Myers, D.S., DeWitt, D.J., Samuel, R.M. : Materialization Strategies in a Column-Oriented DBMS. ICDE'07, 466-475, 2007
2. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K. : Scalable semantic web data management using vertical partitioning. VLDB '07, 411-422, 2007
3. Abadi, D.J., Madden, S., Ferreira, M. : Integrating compression and execution in column-oriented database systems. SIGMOD '06, 671–682, 2006.
4. Brickley, D., Guha, R.V.: RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation (Feb. 2004) `http://www.w3.org/TR/rdf-schema/`
5. Broekstra, J., Kampman, A., Van Harmelen, F. : Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. ISWC'02. 54–68 2002
6. Chong, E.I., Das, S., Eadon, G., Srinivasan, J. : An efficient SQL-based RDF querying scheme. VLDB'05. 1216–1227, 2005
7. Copeland, G.P., Khoshafian, S.N. : A decomposition storage model. SIGMOD'85, 268–279, 1985
8. Curé, O.: Semi-automatic Data Migration in a Self-medication Knowledge-based System. Wissensmanagement'05, 323-329
9. Guo Y., Pan Z., Heflin J. : LUBM: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005
10. Harris, S., Gibbins, N. : 3Store Efficient bulk RDF storage. PSSS'03, 1–20, 2003
11. Hayes, P.: RDF Semantics (Feb. 2004). `http://www.w3.org/TR/rdf-mt/`
12. Kolas, D., Emmons, I., Dean, M. : Efficient Linked-list RDF Indexing in Parliament. SSWS'09, 17-32, 2009
13. Prud'hommeaux, E., Seaborn A.: SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`
14. 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
15. 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
16. Weiss, C., Karras, P., Bernstein, A. : Hexastore : sextuple indexing for semantic web data management. VLDB'08, 1008-1019, 2008
17. Wilkinson, K. : Jena property table implementation. SSWS'06, 2006
18. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D. : Efficient RDF Storage and Retrieval in Jena2. SWDB'03, 131-150, 2003