OntoDBench: Novel Benchmarking System for Ontology-Based Databases

Stéphane Jean, Ladjel Bellatreche, Géraud Fokou, Mickaël Baron, and Selma Khouri

LIAS/ISAE-ENSMA and University of Poitiers
BP 40109, 86961 Futuroscope Cedex, France
{jean,bellatreche,fokou,baron,selma.khouri}@ensma.fr

Abstract. Due to the explosion of ontologies on the web (Semantic Web, E-commerce, and so on) organizations are faced with the problem of managing mountains of ontological data. Several academic and industrial databases have been extended to cope with these data, which are called Ontology-Based Databases ($\mathcal{OBDB}$). Such databases store both ontologies and data on the same repository. Unlike traditional databases, where their logical models are stored following the relational model and most of properties identified in the conceptual phase are valuated, OBDBs are based on ontologies which describe in a general way a given domain; some concepts and properties may not be used and valuated and they may use different storage models for ontologies and their instances. Therefore, benchmarking $\mathcal{OBDB}$ represents a crucial challenge. Unfortunately, existing $\mathcal{OBDB}$ benchmarks manipulate ontologies and their instances with characteristics far away from real life applications in terms of used concepts, attributes or instances. As a consequence, it is difficult to identify an appropriate physical storage model for the target $\mathcal{OBDB}$, which enables efficient query processing. In this paper, we propose a novel benchmarking system called *OntoDBench* to evaluate the performance and scalability of available storage models for ontological data. Our benchmark system allows : (1) evaluating relevant characteristics of real data sets, (2) storing the dataset following the existing storage models, (3) expressing workload queries based on these models and (4) evaluating query performance. Our proposed ontology-centric benchmark is validated using the data sets and workload from the Lehigh University Benchmark (LUBM).

1 Introduction

Ontologies are used to explicit the semantics of a domain through conceptual, formal and consensual models. These nice characteristics have been used in many domains such as e-commerce, engineering or environments to solve a wide range of problems such as information integration, natural language processing or information retrieval. With the widespread of ontologies, the quantity of data described by ontologies, called *ontology-based data*, has quickly increased and as a consequence, the development of scalable solutions represents a crucial challenge. The database technology was one of relevant solutions. The obtained databases are called *ontology-based databases* ($\mathcal{OBDB}$). They aim at storing both ontologies and the data they describe in the same repository. Several commercial and academic database management system (DBMS) have been extended with features designed to manage and query these *mountains of*

ontology-based data [1–10]. Contrary to traditional databases, where only logical representations (usually in the relational format) of the schema is represented in the target DBMS, $\mathcal{OBDB}$ have two entities to be managed: instances and the ontology describing their senses. By deeply examining $\mathcal{OBDB}$, we realize that they differ according to: **(i)** the used *ontological formalisms* to define the ontology like RDF, RDFS, OWL, PLIB, FLIGHT, etc. **(ii)** The *storage schema* of the ontology and of the data model. We distinguish three main *relational* representations: *vertical*, *binary* and *horizontal* (illustrated in Figure 2). Vertical representation stores data in a unique table of three columns *(subject, predicate, object)* [4]. In a binary representation, two-column tables are used for each property [11]. Horizontal representation translates each class as a table having a column for each property of the class [7]. **(iii)** The *architecture* dedicated to store all information (Figure 1). Systems like Oracle [4], use the same architecture of traditional databases with two parts: *data schema part* and the *system catalog part*. In systems like IBM Sor [2], the ontology model is separated from its data which gives an architecture with three parts: the ontology model, the data schema and the system catalog. Systems like Ontodb [7] consider an architecture with *four parts*, where a new part called *meta-schema* is added as a system catalog for the ontology part.

This diversity pushes us to formalize an $\mathcal{OBDB}$ by a 6-tuple as follows:
$< \mathcal{O}, \mathcal{I}, \mathcal{P}op, \mathcal{SMIM}, \mathcal{SMI}, \mathcal{A}r >$

- $\mathcal{O}$: the used ontology. It is also defined by a 4-tuple: $< C, R, Ref, F >$, where:
  - C: denotes the set of *Concepts* of the ontology (atomic concepts and concept descriptions). If we consider the ontology used by the LUBM benchmark [12], the set of concepts are C ={*Person, Student, Employee, Dean, TeachingAssistant, Organization, Program, University, Work, Course, Unit, Stream, GraduateCourse*, ...}
  - R: denotes *Roles* (relationships) of the model. Roles can be relationships relating concepts to other concepts, or relationships relating concepts to data-values (like Integers, Floats, etc). The roles of the LUBM are {*author, member, degreeFrom, masterDegreeFrom, tekeCourse, ...*}.
  - Ref : $C \rightarrow$ (Operator, Exp(C,R)). Ref is a *function* defining terminological axioms of a description logic (TBOX). Operators can be inclusion ($\sqsubseteq$) or equality ($\equiv$). $Exp(C, R)$ is an expression over concepts and roles using constructors of description logics such as *union, intersection, restriction*, etc. (e.g., Ref(Student)$\rightarrow$($\sqsubseteq$, Person $\sqcap$ $\forall$takesCourse(Person, Course))).
  - F: represents the *formalism* (RDF, OWL, etc.) in which the ontology O is described. The formalism used in the LUBM is OWL.
- $\mathcal{I}$: presents the *instances* or ontology-based data.
- $\mathcal{P}op$: C $\rightarrow 2^I$ is a *function* that relates each ontological concept to its instances.
- $\mathcal{SMIM}$: is the *Storage Model* of the ontology.
- $\mathcal{SMI}$: is the *Storage Model* of the instances (binary, horizontal or vertical).
- $\mathcal{A}r$: is the *architecture* model of the the target DBMS supporting the $\mathcal{OBDB}$ (Figure 1).

The diversity of $\mathcal{OBDB}$ in terms of storage models, the used formalisms, the target architectures, etc. represents a big opportunity for researchers to work on the development

of benchmarks. This situation is more challenging than the one faced by existing benchmarks for traditional databases (TPC-C: www.tpc.org) and data warehousing (TPC-H, Star Schema Benchmark [13], etc.), where only one entity representing the instances stored in one format is manipulated. Besides we have already lived the same situation when developing benchmarks for object oriented databases and XML databases, where the target model implementing the instances may differ from database to another (OO7 [14], XML benchmarks (XOO7 [15]).

Several existing benchmarks gave the possibility to evaluate the scalability of $\mathcal{OBDB}$ [12, 16–18]. They contribute in giving hints to a database administrator (DBA) in choosing her/his adequate representation for ontology-based data. But, a supplementary effort needs to be performed by DBA to check whether the benchmark dataset and workload are similar to those present in her/his application. This checking represents a hard task since ontology-based data can be rather structured like relational data or completely unstructured. Moreover, Duan et al. [19] have recently raised doubts over the predictions of these benchmark experiments in realistic scenarios. Indeed they have shown, through the definition of a metric called coherence, that while real datasets range from unstructured to structured data, existing benchmark datasets are mostly high-structured. As a consequence, it is still unclear which database representations (storage) should be used for a given dataset and workload.

Instead of defining a dataset and workload conform to the target application, we propose in this paper an alternative benchmarking system called *OntoDBench*, to benchmark $\mathcal{OBDB}$. It is based on a benchmarking system offering DBA the possibility to estimate the *structuredness* of its dataset and test the scalability of the three main representations of ontology-based data on its real datasets and workload. Then, according to its functionality and scalability needs, DBA may choose the adequate $\mathcal{OBDB}$. *OntoDBench* supports the following functionalities: (1) the evaluation of the characteristics of the real datasets, (2) the identification of the relevant storage models ($\mathcal{SMIM}$ and $\mathcal{SMI}$) for a given $\mathcal{OBDB}$, (3) the expression of the workload queries according to the three representations and (4) the evaluation of the queries performance. To validate the correctness of the benchmarking system, we apply it to the LUBM dataset and workload of queries which has been heavily tested in the literature. Our idea is to check whether benchmarking system gives the results that have been previously established on this dataset. Note that our system is generic in the sense that it is able to evaluate any dataset and workload of queries.

The reminder of this paper is structured as follows. In Section 2 we present the state of the art on the representation of ontology-based data in relational databases. Then we review existing benchmarks and show their limitations by introducing a set of ontology-based data metrics. These limitations lead us to propose a new system for benchmarking $\mathcal{OBDB}$. It is based on a benchmarking system described in Section 3. Section 4 validates our proposal on the LUBM Benchmark. Finally, the paper concludes in Section 5 with a summary of our contributions and a discussion of open issues.

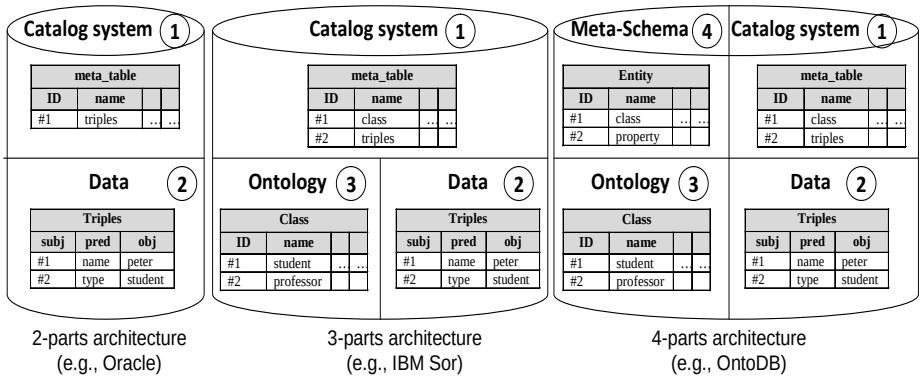

Fig. 1. Different Architectures

2 Background and State of The Art

2.1 Database Representation for Ontology-Based Data

Currently most proposals use relational databases to store ontology-based data. Three main storage layouts have been followed. They are illustrated in Figure 2 based on the LUBM benchmark data [12].

Vertical representation (Figure 2 (a)):

As most ontology-based data are represented in the RDF format, the direct translation into a database schema consists in using a triple table `(subject, predicate, object)`. Since URI are long strings, additional tables may be used to store only integer identifier in the triple table. This approach has initially been followed in Jena [3], 3store [1] and Virtuoso [9]. More recently, Oracle [4] introduces an RDF management platform based on this approach. The main drawback of this approach is that it requires many self-joins over the triple table even for simple queries (shown in section 3.3).

Binary representation (Figure 2 (b)):

An alternative representation to the triple table consists in using a binary (two-columns) table for each property of the dataset. Thus, for a property `P`, a corresponding table `P(subject, object)` stores the values (`object`) of subjects (`subject`) for the property `P`. This representation includes a table `TYPE` to record the type of each subject. An alternative solution consists in using a unary table for each class of the ontology. This approach has been followed in [5, 6, 11] including the systems Sesame and DLDB. Compared to the vertical representation, this approach requires less joins but can still require many joins for query involving many properties.

Horizontal representation: (Figure 2 (c)[1])

To avoid the many joins required with the vertical and binary representation, the horizontal representation consists in representing ontology-based data in a relational way. Thus, for each class `C` of the ontology, a table $C(p_1, \ldots, p_n)$ is defined where $p_1, \ldots, p_n$

are the single-valued properties used at least by one instance of the class. Multi-valued properties are represented by a two-column table such as in the binary representation. This approach is followed in OntoDB [7], OntoMS [8] and Jena2 [16]. Since all subjects in a table do not necessarily have a value for all properties of the table, this representation can be sparse which can impose performance overhead. Moreover, if the ontology is mostly composed of multi-valued properties, this approach is similar to the binary representation.

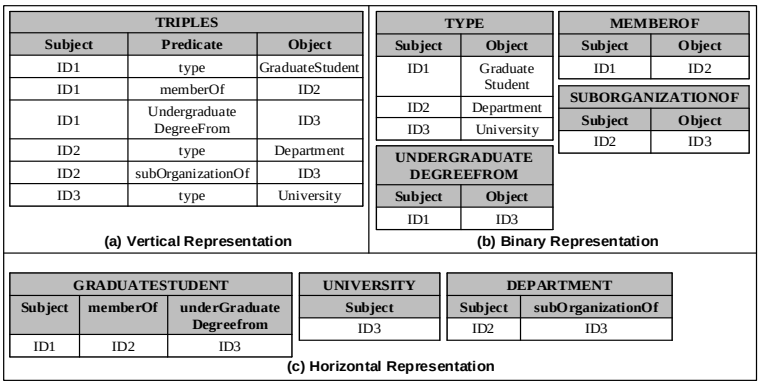

Fig. 2. Example of the three main representations of ontology-based data

As we have seen, three main representations are used to store ontology-based data. The scalability of these three representations has been evaluated in several benchmarks presented in next sections.

2.2 Benchmarking Ontology-Based Database

In this section, we review five *OBDB* benchmarks: *the LUBM Benchmark*, *the Barton Library Benchmark*, *the Berlin SPARQL Benchmark*, *the SP2Bench Benchmark* and *the DBpedia SPARQL Benchmark*.

The LUBM Benchmark [12] is composed of several datasets on a University domain with classes such as `UndergraduateStudent` or `AssistantProfessor` and 14 test queries. Each dataset is named `LUBM(N,S)` where `N` is the number of Universities and `S` is a seed value used for the generation of data. Five sets of test data are generally used: LUBM(1, 0), LUBM(5, 0), LUBM(10, 0), LUBM(20, 0), and LUBM(50,0) (up to seven millions triples) but the generator could also be used to generate more data. This benchmark was used to conduct an evaluation of mainly two *OBDB*s: DLDB [6] and Sesame [5] which were each better at different queries.

The Barton Library Benchmark [17] is composed of a dataset on a Library domain with classes such as `Text` or `Notated Music`. This dataset contains approximately 50 millions of triples. Seven queries were associated to this benchmark in [11]. Experiments

were run to compare the three main representations of RDF data presented in the previous section. The binary representation was clearly better than the vertical representation and was always competitive to the horizontal representation.

The Berlin SPARQL Benchmark [18] is composed of a dataset on an e-commerce domain with classes such as `Product` or `Vendor` and 12 test queries. The aim of this benchmark was to test a real sequence of operations performed by a human user. This benchmark was used to do experiments on $\mathcal{OBDB}$s such as Jena [3], Sesame [5], Virtuoso [9] and OWLIM [10].

The SP2Bench Benchmark [19] is composed of a dataset based on DBLP with classes such as `Person` or `Inproceedings` and 12 test queries. The aim of this benchmark was to include a variety of SPARQL features not supported in previous benchmark. The experiments conducted on $\mathcal{OBDB}$s, including Sesame [5] and Virtuoso [9], identified scenarios where the binary representation was slower than the vertical representation.

The DBpedia SPARQL Benchmark [20] is composed of a dataset based on DBpedia with 25 SPARQL query templates. The aim of this benchmark was to use real RDF dataset instead of generated one. This benchmark was used to do experiments on Virtuoso[9], Jena [3] and OWLIM [10]. The obtained results were more diverse and indicate less homogeneity than what was suggested by other benchmarks.

As we can see, many benchmarks have been proposed for benchmarking $\mathcal{OBDB}$s. They target different domains and evaluate different $\mathcal{OBDB}$s. For a DBA that needs to choose an $\mathcal{OBDB}$, the experiments conducted in these benchmarks are not easy to use as they do not use the same data and thus are often contradictory. As stated in [12], "one must determine if the benchmark is similar enough to a desired workload in order to make predictions about how well it will perform in any given application". Recently, Duan et al. [21] have introduced a set of metrics to estimate this similarity.

2.3 Ontology-Based Data Metrics

Usually, a dataset is described by basic metrics such as the number of triples, subjects, predicates, objects, or classes and detailed statistics such as the average outdegree of the subjects (i.e., the average number of properties associated with a subject) or average indegree of the objects (i.e., the average number of properties associated with an object). However these basic metrics do not characterize the structuredness of a dataset.

Intuitively the structuredness of a dataset is linked to the number of `NULL` values (i.e, values not defined) in the dataset. This number can be computed with the following formula:

$$\#NULL = (\sum_{C} |P(C)| \times |I(C,D)| - Nt(D))$$

- $|P(C)|$ is the number of properties of the class C;
- $|I(C,D)|$ is the number of instances of C in the dataset D;
- $|Nt(D)|$ is the number of triples of the dataset D.

This number of `NULL` can be computed for each class to evaluate the structuredness of each class. This metrics is called *coverage* of a class C in a dataset D, denoted $CV(C,D)$.

$$CV(C,D) = \frac{\sum_{p \in P(C)} OC(p, I(C,D))}{|P(C)| \times |I(C,D)|}$$

– $OC(p, I(C,D))$ is the number of occurrences of a property p for the C instances of the dataset D.

To illustrate the coverage metric lets consider a class that has 2 properties ($P(C)$) and 4 instances ($I(C,D)$). If this class had a perfect coverage, all 4 instances would have a value for the 2 properties (8 triples). Now, if look at the triples and see that the 4 instances have only a value for the first property (4 triples) then the coverage of the class is 0.5 (4/8).

The coverage metric is only defined on a class and do not characterize the whole dataset. Yet a class can be more or less important. If we denote τ the set of classes in the dataset, this weight WT is computed by:

$$WT(CV(C,D)) = \frac{|P(C)| + |I(C,D)|}{\sum_{C' \in \tau(|P(C')| + |I(C',D)|)}}$$

This formula gives higher weight to types with more instances and with a larger number of properties. The weight of a class combined with the coverage metric can be used to compute the structuredness of a dataset called *coherence*. The coherence for dataset D composed of the classes τ, denoted $CH(\tau, D)$ is:

$$CH(\tau, D) = \sum_{C \in \tau} WT(CV(C,D)) \times CV(C,D)$$

Using the coherence metrics Duan et al.[21] showed that benchmark datasets are very limited in their structuredness and are mostly well-structured. In comparison, real datasets range from unstructured to structured data. As a consequence, it is most unlikely that a DBA can use results of existing benchmarks to predict the behaviour of an $\mathcal{OBDB}$ if he uses it to manage its own dataset and workload of queries. In response to this limitation, Duan et al.[21] introduced a benchmark generator which takes as input the coherence of the dataset to be generated. This approach has two limitations: (1) it is difficult to do experiments for the whole spectrum of structuredness. Thus a DBA has to do its own experiments generating a dataset with the desired coherence, loading it in $\mathcal{OBDB}$s and executing queries and (2) the generated dataset is not associated to queries that are similar to the real workload. Again the DBA has to define queries on the generated dataset which are quite similar to her/his real application (same selectivity factors, hierarchies, etc.). This task is not always easy. To reduce the complexity of this task, we propose a novel benchmarking system that can be used to test the scalability of the real dataset and workload of queries according the different storage models of ontology-based data. This benchmarking system is detailed in next section.

3 Our Benchmarking System: OntoDBench

3.1 Dataset Storage

The first step of our benchmark system consists in storing the dataset according to the three main database representations. As in [11], we chose to do a direct *PostgreSQL*

representations instead of using specific $\mathcal{OBDB}$s. This approach has the advantage to give us a direct interaction with database and a control of the queries supported. Conversely, the implemented representations may not support all the specific optimization provided by an $\mathcal{OBDB}$. As we will see in the conclusion, this open issue is part of our future work.

Our **PostgreSQL** implementation of the vertical representation contains the triple table as well as a table that maps URL to integer identifier. As this is mostly done in most $\mathcal{OBDB}$s, three B+ tree indices are created on the triple table: one clustered on (subject, property, object), two unclustered on (property, object, subject) and (object, subject, property). The binary representation contains one two-column table per property. Each table has a clustered B+ tree index on the subject, and an unclustered B+ tree index on the object. Finally, the horizontal representation consists of a table for each class and one for each multi-valued properties. We distinguished single-valued and multi-valued properties by looking at the dataset. In another dataset, this distinction could also be made with schema information (e.g., the cardinality constraint `maxCardinality` of OWL). No specific index (except on the primary key) is defined on the tables corresponding to classes. The table for multi-valued properties is indexed as in the binary representation.

The tables of each representation must be loaded with the dataset. This process is achieved **(1)** by converting all the dataset in the N-Triples format since this format maps directly to the vertical representation, **(2)** inserting each triple in the vertical representation and (3) loading the dataset in the binary and horizontal representations directly from the vertical representation (which was more efficient that reading again the whole file). The conversion in the N-Triples format is done with the Jena API. However, since real dataset can be pretty huge (e.g., the size of the UniProt dataset [22] is approximately 220 GB), the conversion of this dataset could not be done in main-memory. To solve this problem, we do a segmentation of the input file. This segmentation has two advantages: **(1)** huge datasets, decomposed in several files, can be read and **(2)** the loading process can be executed with a multithreaded program (one thread for each file). The graphical interface used for steps **(1)** and **(2)** is presented in Figure 3.

The step (3) consists in loading the binary and horizontal representation from the vertical representation. As the dataset is already in the database, this process can be done directly on the database using stored procedures encoding algorithms 1.1 and 1.2[2]. The graphical interface uses for steps (3) is presented in top of Figure 5.

Algorithm 1.1. Loading the binary representation

Input: $T(s, p, o)$: The triple table;
begin
  foreach $t \in T$ **do**
    $INSERT\ INTO\ t.p\ VALUES\ (t.s,\ t.o)$;
end

[2] For readability, we simplify these algorithms but in the implementation they use less queries (with self-joins) to optimize the processing

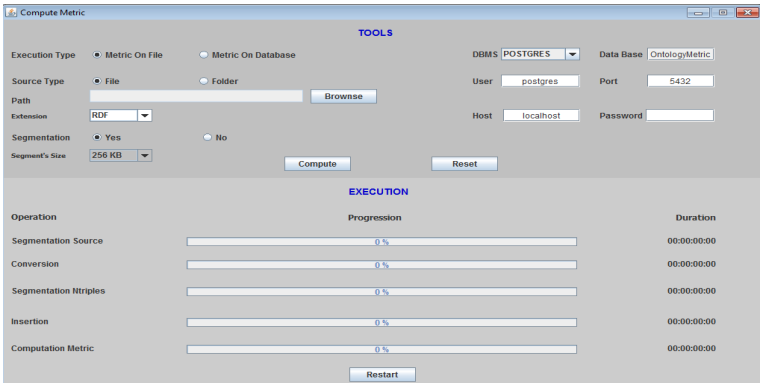

Fig. 3. Interface to load the dataset in the triple table and compute metrics

Algorithm 1.2. Loading the horizontal representation

Input: $T(s, p, o)$: The triple table;

begin

  $C = SELECT\ o\ FROM\ T\ WHERE\ p =' type'$;

  foreach $c \in C$ **do**

    $I_c = SELECT\ s\ FROM\ T\ WHERE\ o =' c'$;

    foreach $i \in I_C$ **do**

      $r : tuple$;

      $T_i = SELECT\ *\ FROM\ T\ WHERE\ s =' i'$;

      foreach $ti \in T_i$ **do**

        if *ti.p is single-valued* **then**

          $r.p = ti.o$;

        else

          $INSERT\ INTO\ ti.p\ VALUES\ (ti.s,\ ti.o)$;

      $INSERT\ INTO\ c\ VALUES\ r$;

end

3.2 Metrics Computation

The second step of our benchmarking system consists in computing the metrics of the dataset. This metric can be used to find the relevant benchmarks to the current scenario. Since the data are already in the database, the computation of most basic metrics is computed with a single SQL query. For example, the following query on the triple table T computes the average indegree of the objects:

```
SELECT AVG(p) FROM T WHERE p <> 'type' GROUP BY o
```

The computation of the coverage and coherence is more complex and is implemented with stored procedures. For example, the coverage of a class C is computed with the algorithm 1.3. The metrics are automatically computed once the dataset is loaded. The result is exported in a text file.

Algorithm 1.3. Computing the coverage of a class C

Input: $T(s, p, o)$: The triple table;
Output: Res: The coverage of the class C;
begin
  $I(C, D) = SELECT\ s\ FROM\ T\ WHERE\ o = 'C'\ AND\ p = 'type'$;
  $|P(C)| = SELECT\ COUNT(DISTINCT\ p)\ FROM\ T\ WHERE\ s\ IN\ I(C, D)$;
  $numerator = 0$;
  foreach $pc \in P(C)$ **do**
    $OC(pc, I(C, D)) = SELECT\ COUNT(*)\ FROM\ T\ WHERE\ p = 'pc'\ AND\ s\ IN\ I(C, D)$;
    $numerator = numerator + OC(pc, I(C, D))$;
  $Res = numerator\ /\ P(C) \times I(C, D)$;
end

3.3 Query Rewriting Module

Once the dataset is loaded in the database, the queries need to be translated according to the three representations. We consider queries consisting of a pattern which is matched against the dataset. If $\mathtt{C}_1, \ldots, \mathtt{C}_n$ are ontology classes and $\mathtt{p}_{11}, \ldots, \mathtt{p}_{nn}$ properties, the considered pattern is the following:
$(?\mathtt{id}_1, \mathtt{type}, \mathtt{C}_1)\ (?\mathtt{id}_1, \mathtt{p}_{11}, ?\mathtt{val}_{11}) \cdots (?\mathtt{id}_1, \mathtt{p}_{n1}, ?\mathtt{val}_{n1})\ [\mathtt{FILTER}()]$
$(?\mathtt{id}_2, \mathtt{type}, \mathtt{C}_2)\ (?\mathtt{id}_2, \mathtt{p}_{12}, ?\mathtt{val}_{12}) \cdots (?\mathtt{id}_2, \mathtt{p}_{n2}, ?\mathtt{val}_{n2})\ [\mathtt{FILTER}()]$
...
$(?\mathtt{id}_n, \mathtt{type}, \mathtt{C}_n)\ (?\mathtt{id}_n, \mathtt{p}_{1n}, ?\mathtt{val}_{1n}) \cdots (?\mathtt{id}_n, \mathtt{p}_{nn}, ?\mathtt{val}_{nn})\ [\mathtt{FILTER}()]$

Notice that these queries could be written with most ontology query languages such as SPARQL. Conversely, this pattern does not yet support specific operators (e.g. `OPTIONAL` in SPARQL) but it could be extended with them.

All queries of the LUBM benchmark could be written with this pattern. For example, Q2 is expressed as follows:
$(?\mathtt{x}, \mathtt{type}, \mathtt{GraduateStudent})\ (?\mathtt{x}, \mathtt{memberOf}, ?\mathtt{z})$
$(?\mathtt{x}, \mathtt{undergraduateDegreeFrom}, ?\mathtt{y})$
$(?\mathtt{y}, \mathtt{type}, \mathtt{University})$
$(?\mathtt{z}, \mathtt{type}, \mathtt{Department})\ (?\mathtt{z}, \mathtt{subOrganizationOf}, ?\mathtt{y})$

Our query rewriting module automatically translates this query to SQL queries on the three possible representations.

Query rewriting on the vertical representation:

Let's $\mathtt{T}(\mathtt{s},\mathtt{p},\mathtt{o})$ be the triple table, $\mathtt{Pred}_1$ the conjunction of predicates in the `FILTER` operators and $\mathtt{Pred}_2$ the conjunction of predicates composed of equalities of variables having the same name, then the considered pattern can be translated by the following relational algebra expressions:

$$\begin{aligned}
\mathtt{FROM} \quad &:= \mathtt{T}_{\rightarrow \mathtt{T}_1} \overset{}{\bowtie}_{\mathtt{T}_1.\mathtt{s}=\mathtt{T}_{11}.\mathtt{s}} \mathtt{T}_{\rightarrow \mathtt{T}_{11}} \ldots \mathtt{T}_{\rightarrow \mathtt{T}_{1n-1}} \overset{}{\bowtie}_{\mathtt{T}_{1n-1}.\mathtt{s}=\mathtt{T}_{1n}.\mathtt{s}} \mathtt{T}_{\rightarrow \mathtt{T}_{1n}} \times \\
&\quad \ldots \\
&\quad \mathtt{T}_{\rightarrow \mathtt{T}_n} \overset{}{\bowtie}_{\mathtt{T}_n.\mathtt{s}=\mathtt{T}_{n1}.\mathtt{s}} \mathtt{T}_{\rightarrow \mathtt{T}_{n1}} \ldots \mathtt{T}_{\rightarrow \mathtt{T}_{nn-1}} \overset{}{\bowtie}_{\mathtt{T}_{nn-1}.\mathtt{s}=\mathtt{T}_{nn}.\mathtt{s}} \mathtt{T}_{\rightarrow \mathtt{T}_{nn}} \\
\mathtt{WHERE} \quad &:= \sigma_{\mathtt{T}_1.\mathtt{p}='\mathtt{type}' \wedge \mathtt{T}_1.\mathtt{o}='\mathtt{C}'_1 \wedge \ldots \wedge \mathtt{T}_n.\mathtt{p}='\mathtt{type}' \wedge \mathtt{T}_n.\mathtt{o}='\mathtt{C}'_n \wedge \mathtt{pred}_1 \wedge \mathtt{pred}_2}(\mathtt{FROM}) \\
\mathtt{SELECT} \quad &:= \pi_{\mathtt{t}_1.\mathtt{s},\ldots\mathtt{t}_n.\mathtt{s},\mathtt{t}_{11}.\mathtt{o},\ldots,\mathtt{t}_{nn}.\mathtt{o}}(\mathtt{WHERE})
\end{aligned}$$

As an example, Figure 4 (a) presents the SQL query corresponding to the query 2 of LUBM. For simplicity's sake, this query only projects distinct variables of Q2.

Query rewriting on the binary representation:

Let's $\mathtt{P}(\mathtt{s},\mathtt{o})$ be the table corresponding to property `P`, and $\mathtt{Pred}_1$, $\mathtt{Pred}_2$ as defined previously, then the considered pattern can be translated by the following relational algebra expressions:

$$\begin{aligned}
\mathtt{FROM} \quad &:= \mathtt{Type}_{\rightarrow \mathtt{T}_1} \overset{}{\bowtie}_{\mathtt{T}_1.\mathtt{s}=\mathtt{P}_{11}.\mathtt{s}} \mathtt{P}_{11} \ldots \mathtt{P}_{1n-1} \overset{}{\bowtie}_{\mathtt{P}_{1n-1}.\mathtt{s}=\mathtt{P}_{1n}.\mathtt{s}} \mathtt{P}_{1n} \times \\
&\quad \ldots \\
&\quad \mathtt{Type}_{\rightarrow \mathtt{T}_n} \overset{}{\bowtie}_{\mathtt{T}_n.\mathtt{s}=\mathtt{P}_{n1}.\mathtt{s}} \mathtt{P}_{n1} \ldots \mathtt{P}_{nn-1} \overset{}{\bowtie}_{\mathtt{P}_{nn-1}.\mathtt{s}=\mathtt{P}_{nn}.\mathtt{s}} \mathtt{P}_{nn} \\
\mathtt{WHERE} \quad &:= \sigma_{\mathtt{T}_1.\mathtt{o}='\mathtt{C}'_1 \wedge \ldots \wedge \mathtt{T}_n.\mathtt{o}='\mathtt{C}'_n \wedge \mathtt{pred}_1 \wedge \mathtt{pred}_2}(\mathtt{FROM}) \\
\mathtt{SELECT} \quad &:= \pi_{\mathtt{t}_1.\mathtt{s},\ldots\mathtt{t}_n.\mathtt{s},\mathtt{p}_{11}.\mathtt{o},\ldots,\mathtt{p}_{nn}.\mathtt{o}}(\mathtt{WHERE})
\end{aligned}$$

As an example, Figure 4 (b) presents the SQL query corresponding to the query 2 of LUBM.

Query rewriting on the horizontal representation:

Let's $\mathtt{C_i}(\mathtt{s},\mathtt{p_{i1}},\ldots,\mathtt{p_{im-1}})$ be the table corresponding to the class $\mathtt{C_i}$ where $\mathtt{p_{i1}},\ldots,\mathtt{p_{im-1}}$ are single-valued properties. If $\mathtt{p_{im}},\ldots,\mathtt{p_{in}}$ are multi-valued properties of $\mathtt{C_i}$ then each multi-valued property `P` has a corresponding table `P(s,o)`. If we use the notations $\mathtt{Pred}_1$, $\mathtt{Pred}_2$ as defined previously, then the considered pattern can be translated by the following relational algebra expressions:

$$\begin{aligned}
\mathtt{FROM} \quad &:= \mathtt{C}_1 \overset{}{\bowtie}_{\mathtt{C}_1.\mathtt{s}=\mathtt{P}_{1m}.\mathtt{s}} \mathtt{P}_{1m} \ldots \mathtt{P}_{1n-1} \overset{}{\bowtie}_{\mathtt{P}_{1n-1}.\mathtt{s}=\mathtt{P}_{1n}.\mathtt{s}} \mathtt{P}_{1n} \times \\
&\quad \ldots \\
&\quad \mathtt{C}_n \overset{}{\bowtie}_{\mathtt{C}_n.\mathtt{s}=\mathtt{P}_{nm}.\mathtt{s}} \mathtt{P}_{nm} \ldots \mathtt{P}_{nn-1} \overset{}{\bowtie}_{\mathtt{P}_{nn-1}.\mathtt{s}=\mathtt{P}_{nn}.\mathtt{s}} \mathtt{P}_{nn} \\
\mathtt{WHERE} \quad &:= \sigma_{\mathtt{pred}_1 \wedge \mathtt{pred}_2}(\mathtt{FROM}) \\
\mathtt{SELECT} \quad &:= \pi_{\mathtt{C}_1.\mathtt{s},\mathtt{C}_1.\mathtt{p}_1\ldots\mathtt{C}_1.\mathtt{P}_{m-1},\mathtt{P}_{1m}.\mathtt{o},\ldots,\mathtt{P}_{1n}.\mathtt{o},\ldots,\ \mathtt{C}_n.\mathtt{s},\mathtt{C}_n.\mathtt{P}_1\ldots\mathtt{Cn}.\mathtt{P}_{m-1},\mathtt{P}_{nm}.\mathtt{o},\ldots,\mathtt{P}_{nn}.\mathtt{o}}(\mathtt{WHERE})
\end{aligned}$$

As an example, Figure 4 (c) presents the SQL query corresponding to the query 2 of LUBM.

3.4 Benchmarking

With the previous query rewriting module, the workload under test can be executed on the three main database representations for ontology-based data. The database buffers can have an influence on the query performance. Indeed, the first execution of a query

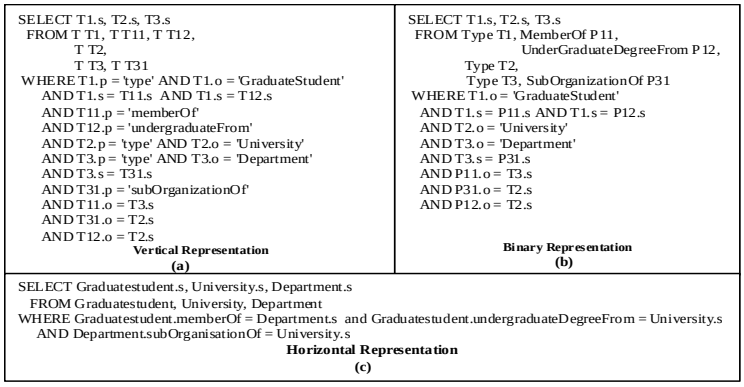

Fig. 4. Query rewriting of the Q2 query of LUBM

is usually slower that the next executions due to the caching of data. As a consequence our benchmarking module takes as input (the graphical user interface is presented on the bottom of Figure 5) the number of times the queries have to be executed. For a given representation R, this process is realized by the algorithm 1.4. All the data of the results Res are automatically exported in an excel file so that the DBA can, if necessary, represent them as graphs.

Algorithm 1.4. Benchmarking the workload queries

Input: Q: The set of m queries;
n: number of executions of each query;
R: representation;
Output: $Res[m, n]$: The set of query response time;
begin
  foreach $Q_i \in Q$ **do**
    $Q'_i = Rewrite(Q_i, R)$;
    foreach $j \in 1 \ldots n$ **do**
      $Res[i, j] = responseTime(Q'_i, R)$;
end

4 Experimental Validation

To validate our proposal we have done a complete implementation of the benchmarking system (the source code is available at http://lisi-forge.ensma.fr/OntologyMetric.zip). We used JAVA for the graphical user interface and PostgreSQL as a database storage. A screenshot of the interface used to load data and compute metrics is presented in Figure 3. Moreover, we run several experiments to test the correctness of our benchmarking

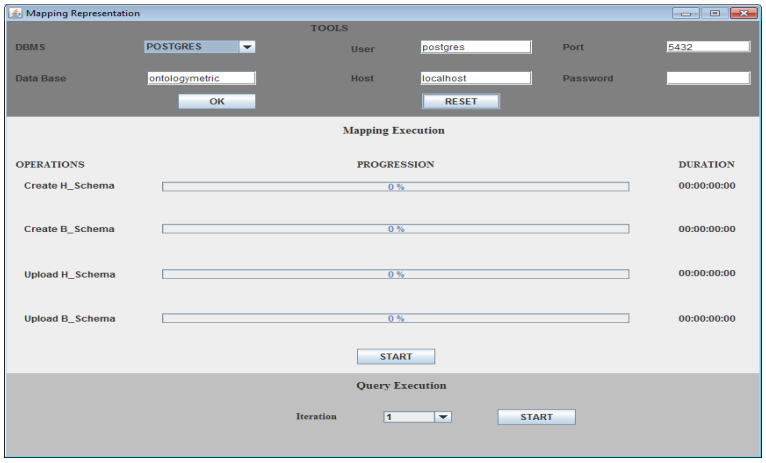

Fig. 5. Interface to load the dataset according to the vertical and horizontal representations and execute workload queries

system. We claim that if it computes the correct metrics of a dataset and provides the correct query performance for a given workload on the three representations we conclude that it is valid. Thus, to test our benchmarking system we need a dataset and a workload of queries that has been heavily tested. As a consequence, we chose the LUBM dataset and its associated 14 queries. To check that our benchmarking system provides the correct answer to each query and that we compute the correct metrics of the dataset, we have used a dataset generated by LUBM (1,0) [3]. These experiments were run on PostgreSQL 8.2 installed on a standard Intel Core i5 750 2.67 Ghz 4GB of RAM desktop machine (Dell). We also run experiments on LUBM(20, 0) and LUBM(50,0) to check that our benchmarking system can handle bigger dataset. The results obtained are described in the following sections.

4.1 Metrics of the LUBM dataset

Table 1 presents the basic metrics computed from the dataset. LUBM(1,0) generates a rather small dataset composed of approximately 1K triples (8MO). The LUBM ontology is composed of 14 classes with 17189 instances. Each instance has an average of approximately 5 values of properties. If we look at the different properties of each class, this dataset is composed of 7244 `NULL` values.

The basic metrics computed does not give a precise idea of the structuredness of the dataset. This structuredness is evaluated by the coherence metric. This metric is computed from the coverage and weight metrics presented in table 2.

[3] Reference query answers for LUBM(1,0) are available at: http://swat.cse.lehigh.edu/projects/lubm/answers.htm

#triples	#subjects	#predicates	#objects	#types	avg indegree	avg outdegree	#NULL
100558	17189	17	13947	14	5,91	4,79	7244

Table 1. Basic metrics of the dataset

Type	Coverage	Weight
AssistantProfessor	100%	1%
AssociateProfessor	100%	1%
Course	100%	5%
Departement	100%	0,10%
FullProfessor	91,20%	1%
GraduateCourse	100%	4%
GraduateStudent	90,22%	10%
Lecturer	100%	1%
Publication	100%	33%
ResearchAssistant	100%	3%
ResearchGroup	100%	1%
TeachingAssistant	100%	2%
UndergraduateStudent	86,79%	33%
University	0,10%	5%

Table 2. Coverage and weight of types

The coverage metric is illustrated on Figure 6. As we can see most classes have a perfect structuredness (instances have a value for each property of the class). There are four exceptions:

- `FullProfessor` has 125 instances with 110 `NULL` (10 properties/instance);
- `GraduateStudent` has 1874 instances with 1467 `NULL` (8 properties/instance);
- `UndergraduateStudent` has 5916 instances with 4689 `NULL` (6 properties/instance);
- `University` has 979 instances with 978 `NULL` (1 property/instance).

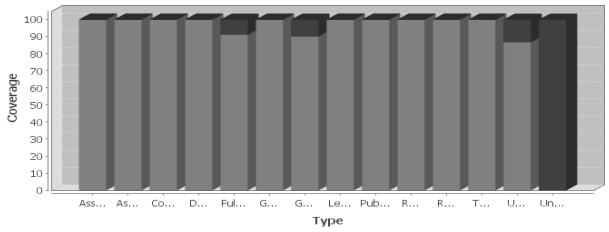

Fig. 6. Coverage of each type of the LUBM dataset

Thus, only the `University` class has a low structuredness due to the fact that this class has only 1 property (`name`) which is valuated by only one instance of this class.

Indeed LUBM(1, 0) generates data for only one university (described by a name) but references many more university that are only identified by an URI.

From the coverage metric we can suspect that this dataset is well-structured. However, we have to take into account the weight (percentage of instances) of each class. Indeed if the `University` class had large number of instances; this dataset could be rather unstructured. The weight metrics is illustrated in Figure 7. As we can see, two classes (`UndergraduateStudent` and `Publication`) have 66% of all instances. Since these two classes have a coverage of approximately 90% it explains that the coherence of the dataset is **89,24%**. This result confirms the one presented in [21] and thus is a good indication that our benchmarking system correctly computes the metrics of a dataset.

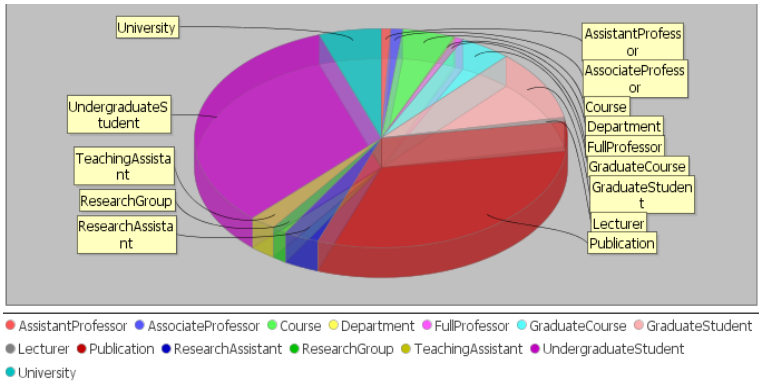

Fig. 7. Weight of each type of the LUBM dataset

4.2 Performance of the LUBM queries on the three representations

The second result that can be obtained from our benchmarking system is the query response time of each query of the workload. Each query was run several times: once, initially, to warm up the database buffers and then four more times to get the average execution time of a query.

Figure 8 presents the obtained *average query response time*. For the readability reason we only present the response time of 9 queries of the LUBM benchmark (Q1 to Q9). These results show that, for all queries, the binary and horizontal representations outperform the vertical representation. This result is caused by two factors: (1) the vertical representation implies many self-joins on the triple table for all LUBM queries. In comparison, these queries involved less joins on the two other representations and on tables that have less tuples. (2) Due to the metrics of the dataset, the two other representations do not suffer from their drawbacks. The dataset contains only 17 properties and most queries do not involve more than 3 properties. Thus, most queries do not involve

many joins in the binary representation. The dataset is well structured and thus, the tables in the horizontal representation are not sparse (except for the table corresponding to `University` but this table is only access by the Q2 query).

The binary and horizontal representations give similar response time. The horizontal representation has a slight advantage for Q7, Q8 and Q9 since they involved only single-valued properties and target classes that have coverage of 100%. The binary representation is better for Q2 and Q4 as these queries involve classes that do not have a perfect coverage (`University` and `FullProfessor`).

These results are similar to the ones obtained in [12, 7]: binary and vertical representations usually outperform the vertical representation for high-structured data. If a DBA put the scalability as the main criteria of choice for an $\mathcal{OBDB}$, our benchmarking system advises her/him to choose an $\mathcal{OBDB}$ that uses a binary or horizontal representation. Of course many other factors can influence the decision of a DBA (e.g., functionalities of the $\mathcal{OBDB}$). One of the most contributions of our benchmarking system is that it gives hints on the scalability criterion.

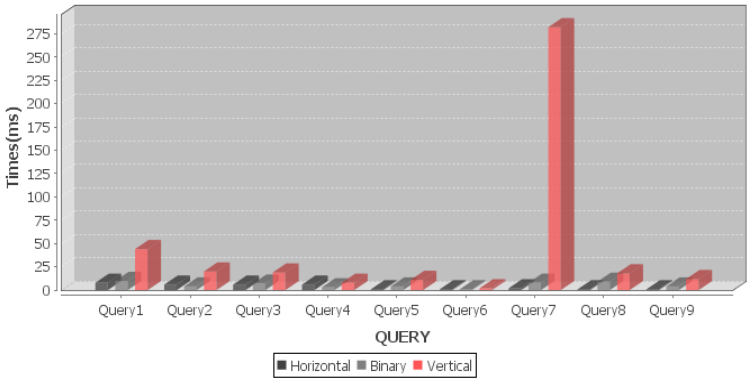

Fig. 8. Average time of query execution for 9 queries of LUBM

The experiments conducted in this section show that our system applied to the LUBM dataset and benchmark provide the same results that have been given in existing benchmarks and thus, is a positive test for validating the correctness of our approach. Compared to existing benchmarks, the interest of our work is that we can do exactly the same experiments on other dataset and workload with the availability of a complete benchmarking system. This contribution is summarized in the next section.

5 Conclusion

The explosion of ontology-based data over the Web pushes commercial and academic DBMS to extend their features to support this new type of data. Notice several research efforts have been undertaken to propose benchmarks in the context of traditional

databases (relational and object-oriented). Ontology-based databases ($\mathcal{OBDB}$) come with new characteristics: (1) they store both ontology-based data and the ontology describing their meanings on the same repository, (2) the concepts and properties in the stored ontology are not always used contrary to traditional databases, (3) ontology and ontology-based data may be stored in various storage layouts (binary, horizontal and vertical) and (4) the target DBMS may have different architectures (2-parts, 3-parts and 4-parts). Existing benchmarks dedicated for $\mathcal{OBDB}$ provide useful hints for the management of very well structured ontology-based data. However, real ontological applications manage unstructured to structured data. In this paper, we propose a benchmarking system *OntoDBench* helping DBA on choosing the relevant components (architecture, storage model, etc.) of their $\mathcal{OBDB}$ that takes into account the data set and workload. Before describing in details this system, we give a generic formalization of an $\mathcal{OBDB}$. Our benchmarking system gives the DBA metrics to evaluate the dataset (including the ontology and their instances) and query performance based on the three storage models. Our benchmarking system full source code and executable programs are available at http://lisi-forge.ensma.fr/OntologyMetric.zip using JAVA for the graphical user interface and PostgreSQL as the DBMS. Our benchmarking system has been evaluated on several LUBM datasets and compared with the existing ones [21].

Our work opens several research issues. $\mathcal{OBDB}$ can use many database optimizations that are not included in our benchmarking system (e.g., partitioning or materialized views). These database optimizations have a huge impact on query performance. Thus we plan to extend our system so that the DBA can compare specific $\mathcal{OBDB}$ representations instead of the generic ones that we have currently implemented. Another perspective concerns the metrics used to characterize a dataset. Currently, the metrics proposed by [21] that we have integrated in our system are only defined on ontology-based data. Yet, queries could also be expressed on ontologies or on both ontologies and data. Thus, it could be interesting to study which metrics can be defined on ontologies (e.g., the depth of the class and/or property hierarchy) are relevant for such queries. Moreover, queries themselves could be characterized by different optimization metrics (e.g., number of joins or selectivity of selections) to help choosing the best database representation according to the dataset and query workload.

References

1. Harris, S., Gibbins, N.: 3store: Efficient Bulk RDF Storage. In: Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03). (2003) 1–15
2. Lu, J., Ma, L., Zhang, L., Brunner, J.S., Wang, C., Pan, Y., Yu, Y.: Sor: a practical system for ontology storage, reasoning and search. In: Proceedings of the 33rd international conference on Very large data bases (VLDB'07). (2007) 1402–1405
3. B.McBride: Jena: Implementing the RDF Model and Syntax Specification. (2001)
4. Wu, Z., Eadon, G., Das, S., Chong, E.I., Kolovski, V., Annamalai, M., Srinivasan, J.: Implementing an Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle. In: Proceedings of the 24th International Conference on Data Engineering (ICDE'08). (2008) 1239–1248

5. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: Proceedings of the 1st International Semantic Web Conference (ISWC'02). (2002) 54–68
6. Pan, Z., Heflin, J.: Dldb: Extending relational databases to support semantic web queries. In: Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03). (2003) 109–113
7. Dehainsala, H., Pierra, G., Bellatreche, L.: OntoDB: An Ontology-Based Database for Data Intensive Applications. In: Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07). (2007) 497–508
8. Park, M.J., Lee, J.H., Lee, C.H., Lin, J., Serres, O., Chung, C.W.: An Efficient and Scalable Management of Ontology. In: Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07). (2007) 975–980
9. Erling, O., Mikhailov, I.: RDF Support in the Virtuoso DBMS. In: Conference on Social Semantic Web (CSSW'07). Volume 113. (2007) 59–68
10. Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., Velkov, R.: OWLIM: A family of scalable semantic repositories. Semantic Web **2**(1) (2011) 1–10
11. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07). (2007) 411–422
12. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Journal of Web Semantics **3**(2-3) (2005) 158–182
13. O'Neil, P., O'Neil, E.J., Chen, X., Revilak, S.: The star schema benchmark and augmented fact table indexing. In: First TPC Technology Conference (TPCTC). (2009) 237–252
14. Carey, M.J., DeWitt, D.J., Naughton, J.F.: The oo7 benchmark. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD). (1993) 12–21
15. Bressan, S., Lee, M., Li, Y.G., Lacroix, Z., Nambiar, U.: The xoo7 benchmark. In: Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web (EEXTT). (2002) 146–147
16. Wilkinson, K.: Jena Property Table Implementation. In: Proceedings of the 2nd International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS'06). (2006) 35–46
17. D. Abadi, A. Marcus, S.M., Hollenbach, K.: Using the Barton libraries dataset as an RDF benchmark. Technical Report MIT-CSAIL-TR-2007-036, MIT (2007)
18. Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. Semantic Web and Information Systems **5**(2) (2009) 1–24
19. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP2Bench: A SPARQL Performance Benchmark. In: Proceedings of the 25th International Conference on Data Engineering (ICDE'09). (2009) 222–233
20. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.C.: DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data. In: Proceedings of the 10th International Semantic Web Conference (ISWC'11). (2011)
21. Duan, S., Kementsietsidis, A., Srinivas, K., Udrea, O.: Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In: Proceedings of the 2011 international conference on Management of data (SIGMOD'11). (2011) 145–156
22. Apweiler, R., Bairoch, A., Wu, C.H., Barker, W.C., Boeckmann, B., Ferro, S., Gasteiger, E., Huang, H., Lopez, R., Magrane, M., Martin, M.J., Natale, D.A., ODonovan, C., Redaschi, N., Yeh, L.S.: Uniprot: the Universal Protein knowledgebase. Nucleic Acids Research **32** (2004) D115–D119