OntoDBench: Interactively Benchmarking Ontology Storage in a Database

Stéphane Jean[1], Ladjel Bellatreche[1], Carlos Ordonez[2], Géraud Fokou[1], Mickaël Baron[1]

[1] LIAS/ISAE-ENSMA, Futuroscope, France
(jean,bellatreche,fokou,baron)@ensma.fr
[2] University of Houston, Houston, U.S.A.
ordonez@cs.uh.edu

1 Introduction

Nowadays, all ingredients are available for developing domain ontologies. This is due to the presence of various types of methodologies for creating domain ontologies [3]. The adoption of ontologies by real life applications generates mountains of ontological data that need techniques and tools to facilitate their storage, management and querying. The database technology was one of these solutions. Several academic and industrial database management systems (DBMS) have been extended with features designed to manage and to query this new type of data (e.g., Oracle [10], IBM Sor [6] or OntoDB [1]). The obtained databases are called *semantic databases* ($\mathcal{SDB}$).

Five main characteristics differentiate $\mathcal{SDB}$ from traditional databases. **(i)** They store both ontologies and their instances in the same repository. **(ii)** Three main storage layouts are candidates for storing ontologies and their instances: vertical (triple table), horizontal (one table by class), binary (one table by property) [8], where each one has its own advantages and drawbacks. **(iii)** $\mathcal{SDB}$ have three main architectures. Systems such as Oracle [10] use the traditional databases architecture with two parts: *data schema part* and the *system catalog part*. In systems such as IBM Sor [6], the ontology is separated from its instances resulting in an architecture with three parts: the *ontology part*, the *data schema part* and the *system catalog part*. OntoDB [1] considers an architecture with *four parts*, where a new part called the *meta-schema part* is added as a system catalog for the ontology part. **(iv)** The ontology referencing $\mathcal{SDB}$ instances may be expressed in various formalisms (RDF, RDFS, OWL, etc.). **(v)** Ontology instances can be rather structured like relational data or be completely unstructured. Indeed, some concepts and properties of an ontology may not be used by a target application. As a consequence, if only a fragment of the domain ontology is used, ontology instances have a lot of *NULL* values. On the contrary, the whole ontology could be used resulting in relational-like ontology instances.

These characteristics make the development of $\mathcal{SDB}$ benchmarks challenging. Several benchmarks exist for $\mathcal{SDB}$ [7, 9, 5]. They present the following drawbacks: **(i)** they give contradictory results since they used datasets and queries with different characteristics. **(ii)** A gap exists between the generated datasets used by existing benchmarks and the real datasets as shown in [2]. **(iii)** The absence of an interactive tool to facilitate the use of those benchmarks. **(iv)** The task of setting all benchmark parameters is *time consuming* for the DBA.

Recently, Duan et al. [2] introduced a benchmark generator to overcome the gap between the generated datasets of existing benchmarks and the real datasets. This benchmark generator takes has input the structuredness of the dataset to be generated. However, this approach has two limitations: **(i)** it is difficult to do experiments for the whole spectrum of structuredness. Thus the DBA has to do its own experiments generating a dataset with the desired structuredness, loading it in $\mathcal{SDB}$ and executing queries and **(ii)** 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 similar to her/his real application (same selectivity factors, hierarchies, etc.). Instead of defining a dataset and workload conform to the target application, we propose an alternative benchmarking system called *OntoDBench* to evaluate $\mathcal{SDB}$. The main difference with previous benchmarking systems is that *OntoDBench* takes as input *the real datasets and workload* of the DBA instead of using a generated dataset and predefined set of queries. *OntoDBench* has two main functionalities. Firstly it evaluates the scalability of the real workload on the three main storage layouts of $\mathcal{SDB}$. Then, according to her/his functionality and scalability requirements, the DBA may choose the adequate $\mathcal{SDB}$. This functionality is based on a rewriting query module that translates input queries according to the different storage layouts and includes ontology reasoning. Secondly, *OntoDBench* offers the DBA the possibility to estimate and modify the characteristics of its input dataset (e.g., structuredness of ontology instances or size of the ontology hierarchy) and workload (e.g., number of joins or selectivity factors of selections). This functionality can be used by the DBA to check whether an existing benchmark (e.g., the DBpedia SPARQL benchmark [7]) uses a dataset and workload similar to those present in her/his application. It can also be used to predict the behavior of storage layouts if ontology data and/or queries change.

2 OntoDBench: Metrics and Demonstration Description

Metrics play a key role in benchmark systems. Usually they are used to generate data with particular characteristics. In *OntoDBench*, they are used both for modifying the real dataset and for checking if the results of an existing benchmark are relevant to the real dataset/workload that must be managed. Since $\mathcal{SDB}s$ store both ontologies and their instances and execute semantic queries, three types of metrics must be considered.
Ontology metrics: they include metrics such as the number of classes, the number of properties by class or the size of class and property hierarchies. The ontology is also characterized by the fragment of Description Logics used (to characterize the complexity of reasoning).
Instance metrics: they include metrics such as the number of instances by class or the average number of properties associated with a subject. We also consider the structuredness of a dataset which is defined in [2] according to the number of `NULL` values in the dataset. This number can be computed with the following formula: $\#NULL = (\sum_{C} |P(C)| \times |I(C,D)| - Nt(D))$, where $|P(C)|$, $|I(C,D)|$ and $|Nt(D)|$ represent the number of properties of the class C, the number of instances of C in the dataset D and the number of triples of the dataset D respectively.
This number of `NULL` can be computed to evaluate the structuredness of each class. This metrics is called *coverage* of a class C in a dataset D, denoted $CV(C,D)$. It is defined as follows: $CV(C,D) = \frac{\sum_{p \in P(C)} OC(p,I(C,D))}{|P(C)| \times |I(C,D)|}$, where $OC(p,I(C,D))$ is the number of occurrences of a property p for the C instances of the dataset D.
A class can be more or less important in a dataset. 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 weights to the 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 of a dataset D composed of the classes τ (denoted $CH(\tau,D)$) is defined by: $CH(\tau,D) = \sum_{C \in \tau} WT(CV(C,D)) \times CV(C,D)$.
Query metrics: the considered semantic queries consist of conjunctive queries composed of selection and join operations. Thus these metrics include characteristics such as the selectivity factors of predicates or the number of join operations in the query.
DataSet Segmentation: loading a *big dataset* in the benchmark repository represents a real difficulty, especially when the size of these data exceeds the main memory. To overcome this problem,

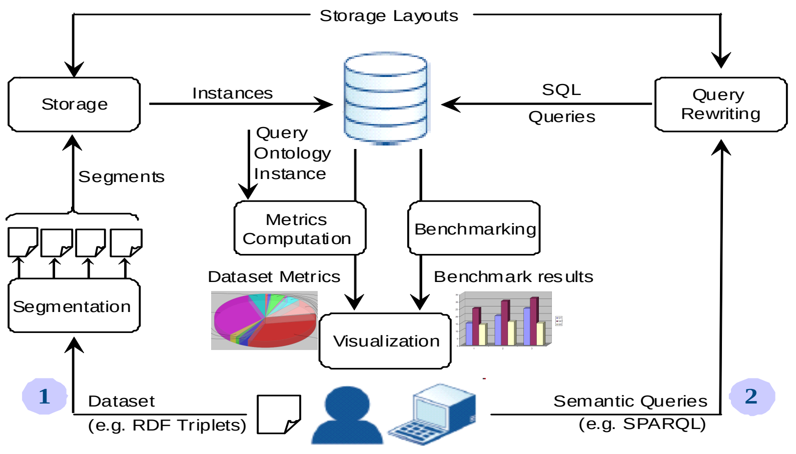

Fig. 1. The Components of our System

we add a new component allowing the DBA to segment the input files. We give the possibility to set the size of segments.

Dataset Storage: this module offers various possibilities to store the incoming dataset segments according to the three storage layouts. The loading process is executed with a multithreaded program (one thread for each segment). This process is achieved by **(1)** converting all the dataset in the N-Triples format since it maps directly to the vertical storage layout, **(2)** inserting each triple in the vertical storage layout and **(3)** loading the dataset in the binary and horizontal storage layouts directly from the vertical storage layout (which was more efficient that reading again the input files). The conversion in the N-Triples format is done with the *Jena API*.

Metrics Computation: the second step of *OntoDBench* 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 done with an SQL query. The computation of the coverage and coherence is more complex and is implemented with stored procedures. The metrics are automatically computed once the dataset is loaded and exported in a text file.

Query Rewriting Module: once the dataset is loaded in the database, the queries need to be translated according to the three storage layouts.

Benchmarking: with the previous query rewriting module, the workload under test can be executed on the three main storage layouts for ontology instances. The database buffers can have an influence on the query performance. Indeed, the first execution of a query is usually slower that the next executions due to the caching of data. As a consequence our benchmarking module takes as input the number of times the queries have to be executed.

Visualization: The benchmarking results (metrics, query processing, etc.) are stored in a text file. The DBA may visualize them via charts, histograms, etc. to facilitate their interpretation and exploitation for reporting and recommending a storage layout. The graphs are generated with the Java JFreeChart API[3].

Reasoning: Our system implements the two main reasoning approaches: (1) database saturation that consists in performing reasoning before query processing and to materialize all the deduced facts and (2) query reformulation that consists in performing reasoning during query processing

[3] www.jfree.org/jfreechart/

by reformulating queries to include all virtual deduced facts. *OntoDBench* offers the DBA the possibility to test these two approaches. For the moment we have implemented the entailment rules of RDFS. For the database saturation approach, we use the PL/pgSQL database programming language to implement the 14 rules of RDFS. For the query reformulation approach, we have implemented the *reformulate algorithm* proposed in [4]. Figure 1 summarizes the different components of our system.

To validate our proposal we have done an implementation of *OntoDBench* (the source code is available at http://www.lias-lab.fr/forge/projects/ontodbench/files). We have used JAVA for the graphical user interface and PostgreSQL as a database storage. A demonstration video summarizing the different services offered by our benchmark is available at: http://www.lias-lab.fr/forge/ontodbench/video.html. The demonstration proposed in this paper consists in using *OntoDBench* on the LUBM dataset with a size ranging from 1K to 6 millions triplets and 14 queries. *OntoDBench* proposes a user friendly interface for the DBA so she/he can choose the size of segments, the storage layouts, etc. We demonstrate the following:

- the loading of huge amount of data by offering *a segmentation mechanism* that partition data in segments of *a fixed size*;
- the possibility to store the loaded data into the target DBMS according to the three main storage layouts (horizontal, vertical and binary);
- the computation of the dataset and workload metrics of the LUBM benchmark. The DBA may easily identify the degree (*high, medium and low*) of structuredness of the dataset. If the DBA is not satisfied with the obtained results, *OntoDBench* offers her/him the possibility to update the dataset to fit with her/his structuredness requirements;
- the query rewriting module of our system. We show the SQL translation of the LUBM queries on the three main storage layouts;
- the benchmarking of each query and the visualization of its result using graphs. The DBA may observe the query processing cost of all queries on the different storage layouts.

References

1. H. Dehainsala, G. Pierra, and L. Bellatreche. OntoDB: An Ontology-Based Database for Data Intensive Applications. In *DASFAA*, pages 497–508, 2007.
2. S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *SIGMOD*, pages 145–156, 2011.
3. C. Garcia-Alvarado, Z. Chen, and C. Ordonez. Ontocube: efficient ontology extraction using olap cubes. In *CIKM*, pages 2429–2432, 2011.
4. F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. View Selection in Semantic Web Databases. *PVLDB Journal*, 5(2):97–108, 2011.
5. Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.
6. J. Lu, L. Ma, L. Zhang, J.-S. Brunner, C. Wang, Y. Pan, and Y. Yu. Sor: a practical system for ontology storage, reasoning and search. In *VLDB*, pages 1402–1405, 2007.
7. M. Morsey, J. Lehmann, S. Auer, and A.-C. Ngonga Ngomo. DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data. In *ISWC*, pages 454–469, 2011.
8. C. Ordonez and P. Cereghini. Sqlem: Fast clustering in sql using the em algorithm. In *SIGMOD*, pages 559–570, 2000.
9. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL Performance Benchmark. In *ICDE*, pages 222–233, 2009.
10. Z. Wu, G. Eadon, S. Das, E. I. Chong, V. Kolovski, M. Annamalai, and J. Srinivasan. Implementing an inference engine for rdfs/owl constructs and user-defined rules in oracle. In *ICDE*, pages 1239–1248, 2008.