WaterFowl: a Compact, Self-indexed and Inference-enabled RDF Store

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

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

Abstract. In this paper, we present a novel approach – called WaterFowl – for the storage of RDF triples that addresses some key issues in the contexts of big data and the semantic web. The architecture of our prototype, largely based on the use of succinct data structures, enables to represent triples in a self-indexed, compact manner and not requiring decompression at query answering time. Moreover, it is adapted to support RDF and RDFS entailment regimes thanks to an optimized encoding of ontology concepts and properties that does not require inferences materialization or extensive query rewriting algorithms. The paper describes the complete architecture of this system and presents some preliminary results obtained from evaluations conducted on our first prototype.

1 Introduction

The emergence of big data forces to face important data management issues: the most predominant ones being scalability, distribution, fault tolerance and low latency query answering. The current trends in handling large volumes of information focus on parallel processing with MapReduce [4] inspired frameworks. We consider that, for at least cost efficiency reasons, this approach may soon not be satisfactory anymore and should be combined with local data compression, *i.e.*, on each machine of a cluster. Hence, one will get the most out of a data center by distributing a dataset over a cluster of machines and by compressing each partition in an clever way. RDF data is totally concerned with this phenomemon partly due to the production of an increasing number of voluminous datasets. In the context of the Semantic Web, handling inferences within query processing adds up to the list of previously cited issues. Moreover, partitioning graph oriented data (process needed in data distribution) is known to be an harder problem in comparison to relational data partitioning. Several works – such as Virtuoso[3] and BigOWLIM[4] – have already tackled the development of such systems. Nevertheless, developing such system is still considered as a challenging and crucial open problem.

[3] http://virtuoso.openlinksw.com/
[4] http://www.ontotext.com/owlim

In this paper, we design a new architecture of RDF database systems that addresses the compression and inference-enabled query answering and evaluate it using a proof of concept prototype. This framework will serve as a building site for remaining issues such as data partitioning and supporting data updates. The foundation of our system consists in a high compression, self-indexed storage structure provided with data retrieving decompression-free operations. By self-indexed, we mean that one can seek and retrieve any portion of the data without accessing the original data itself. Succinct Data Structures (henceforth SDS) provide such properties and are extensively used in our architecture via wavelet trees. The high rate compression obtained from SDS enables the system to keep all the data in-memory and limits latencies associated to Input/Output operations (see Section 5). Based on a preliminary work of Fernández *et al.* called HDT (Header Dictionary Triples) [5] – considered as a first attempt in this direction – we propose to push its inner concept further to its logical conclusion by relying exclusively on bit maps and wavelet trees at all levels of our architecture (see Section 4). Moreover, the used data structures motivate the design of an original query processing solution that integrates efficient optimization and RDFS inferences which were not considered nor in [5] or in [12]. The basic idea is to use a clever encoding of the data that will capture the subsumption relationship of both concepts and properties. Therefore, the encoded data will enclose – without extra cost – both raw data and ontology hierarchies. To efficiently use this encoding, the system will need to perform prefix based rank and select operations [7]. This solution will spare the use of an expensive query rewriting approach [18, 15] or inferences materialization when requesting a given ontology element and all its sub-elements.

This paper is organized as follows. In Section 2, we provide background knowledge on RDF and SPARQL as well as automata-based representation of a dictionary and SDS. Section 3 presents related works in the domains of RDF data management systems and query answering in the presence of inferences. Section 4 details the main components of our architecture. Our proof of concept prototype is evaluated in Section 5.

2 Background

2.1 RDF and SPARQL

Assuming disjoint infinite sets U (RDF URI references), B (blank nodes) and L (literals), a triple (s,p,o) $\in$ (U $\cup$ B) x U x (U $\cup$ B $\cup$ L) is called an RDF triple with s, p and o respectively being the subject, predicate and object. We now also assume that V is an infinite set of variables and that it is disjoint with U, B and L. We can recursively define a SPARQL[5] graph pattern as follows: (i) a triple $gp \in$ (U $\cup$ V) x (U $\cup$ V) x (U $\cup$ V $\cup$ L) is a SPARQL graph pattern, (ii) if gp_1 and gp_2 are graph patterns, then ($gp_1.gp_2$) represents a group of graph patterns that must all match, (gp_1 `OPTIONAL` gp_2) where gp_2 is a set of patterns

[5] http://www.w3.org/TR/rdf-sparql-query/

that may extend the solution enduced by gp_1, and (gp_1 `UNION` gp_2), denoting pattern alternatives, are graph patterns and (iii) if gp is a graph pattern and C is a built-in condition then the expression (gp `FILTER` C) is a graph pattern that enables to restrict the solutions of a graph pattern match according to the expression C. The SPARQL syntax follows the select-from-where approach of SQL queries. The `SELECT` clause specifies the variables appearing in the result set of the query.

In [9], extensions to SPARQL semantics, called entailment regimes, are presented. In this work, we address RDF and RDFS entailment regimes in the context of skolemization as presented in [9], *i.e.*, a syntactic transformation that replaces blank nodes by 'new' names. Most of the inferences we are considering are related to entailment rules proposed in [10]. In the context of query answering, these rules are useful to check satisfiability and rewrite queries. In our system, they are implemented through the use of adapted encodings and data structures which are directly motivated by SDS.

2.2 Automata-based dictionary of non-concept/property elements

The purpose of the dictionary is to support the encoding of URIs, blank nodes and literals encountered as subject, predicate or object of the triples. This encoding takes the form of a key/value pair where the key is a unique integer and the value is either an URI, a blank node or a literal. Our system requires a two-way access to this structure: from keys to values (*i.e.*, to encode the triples and to translate the result set of a query) and from values to keys (*i.e.*, to translate the SPARQL queries from URIs and strings to identifiers as well as to handle SPARQL `FILTER` clauses). Note that this dictionary approach, a first compression strategy, is frequently encountered in triple store solutions but usually, provided without any implementation details. The dictionary implementation adopted in WaterFowl is, depending on the data set, based on either an automaton or a trie. Automata are very efficient data structures for natural languages lexicon representation [16], with efficient time and space complexities. In datasets, the strings associated with subjects, predicates and objects are of two kinds : a set of similar strings (sharing common prefixes - *e.g.*, namespaces) and a set of singletons. The automata can be stored in memory and acts as a reversible mimimal perfect hash function on the set of strings. Considering the set of singletons, automata do not provide a compression gain. Then, one may store the singletons into a flat file on the disk and use a trie in order to bind a given prefix to the position of the corresponding word in the file. With the right file system implementation, this yields to a unique file access per string. Those techniques of minimal perfect hashing with DAWG (Directed Acyclic Word Graphs) or Tries enable us to encode any subject, predicate and object as an integer which will be used in the compressed version of the set of the triples. Futhermore, independently to where they appear (as subject or object) we can use a common encoding due to our layered architecture (see Section 4) – leading to a gain of

space. All related implementations are realized using the C++ template library ASTL[6].

2.3 Succinct Data Structures

The family of SDS uses a compression rate close to theoretical optimum, but simultaneously allowing efficient decompression-free query operations on the compressed data. This property is obtained using a small amount (o(Z) bits where Z corresponds to the theoretical optimum) of extra bits to store extra information. Initially introduced by Jacobson [11] when considering bit vectors, the concept is nowadays extended to wider alphabets. Bit vectors are useful to represent data while minimizing its memory footprint. In its classical shape, a bit vector allows, in constant time, to access and modify a value of the vector. Munro [13] designed an asymptotic optimal version where, in constant time, one can (i) count the number of 1 (or 0) appearing in the first x elements of a bit vector (denoted $rank_b(x)$ with b $\in$ {0,1}), (ii) find the position of the x^{th} occurrence of a bit (denoted $select_b(x)$, b $\in$ {0,1}) and (iii) retrieve the bit at position x (denoted $access(x)$). Naturally, these operations on bit vectors would be of great interest for a wider alphabet. The original solution was provided by Grossi *et al.* [6] and roughly consists in using a balanced binary tree – so-called wavelet tree. The alphabet is splitted into two equal parts. One attributes a 0 to each character of the first part and a 1 to the others. The original sequence is written, at the root of the tree, using this encoding. The process is repeated, in the left subtree, for the subsequence of the original sequence only using characters of the first part of the alphabet and, in the right subtree, for the second part. The process iterates until ending up on singleton alphabet. Roughly, one has provided an encoding of each character of the alphabet. Using rank and select operations on the bit vectors stored in the nodes of the tree, one is able to compute rank and select operations on the original sequence in O(log |alphabet|) by deep traversals of the tree. These operations can be easily adapted to only traverse until a given depth – refered as rank_prefix and select_prefix operations (that will be of great interest for us along with our clever encoding). Wavelet trees have been well studied since then and both space and time efficient implementations are now available (*e.g.*, pointer-free wavelet tree and wavelet matrix of libcds library[7]).

3 Related work

In this section, we consider related works in the fields of indexing RDF datasets, query processing in the presence of inferences and query optimizations. Although the first RDF stores appeared in 2002, *e.g.*, [3], this research field became really active in 2007 starting with the publication of [1]. This paper motivated the development of systems which were not using relational database management

[6] http://astl.sourceforge.net
[7] https://code.google.com/p/libcds/

systems as a storage layer and were considering indexes with more attention. Hence, solutions such as Hexastore [21] and RDF 3X [14] were designed using multiple indexes, respectively 6 and 15, which had a direct impact on the performance of query answering but also on the memory footprint of databases. Matrix Bit loaded [2] is another multiple indexes solution which stores its data into bit matrices. Compared to these systems, our approach proposes a single structure that enables an indexed access on the three components of the triples.

Our approach is inspired from the HDT FoQ [12] solution which mainly focuses on data exchange (and thus on data compression). In fact, WaterFowl brings this approach further to its logical conclusion by using a pair of wavelet trees on the object layer (HDT FoQ uses an adjaceny list for this layer – without justification) and by integrating a complete query processing solution with completeness of RDFS reasoning (*i.e.*, handling any inference using RDFS expressiveness). It is made possible by an adaptation of both the dictionary and the triples structures.

Concerning query processing in the presence of inferences, several approaches have been proposed. Among them, the materialization of all inferences within the data storage solution is a popular one which is generally performed using an off-line forward chaining approach. This avoids query formulation at run-time but is associated with an expansion of the memory footprint. Sesame[8] is a commercial system adopting inference materialization. Another approach consists in performing query rewriting at run time. It guarantees a light memory footprint but it is associated with the possible generation of an exponential number of queries. Presto [18] and Requiem [15] are systems adopting this approach with different algorithms. The encoding of ontology elements, *i.e.*, concepts and properties, performed in our system is related to a third approach presented in [17] and implemented in the Quest system (a relational database management system). The work of Rodriguez-Muro *et al.* relies on integer identifiers modeling the subsumption relationships which are being used to rewrite SQL queries ranging over identifiers intervals, *i.e.*, specifying boundaries over indexed fields in the WHERE clause of a SQL query. In comparison, our work focuses on the sharing of common prefixes in the encoding of the identifiers (see Section 4.1). This approach allows us to rewrite the queries in terms of rank_prefix and select_prefix operations, *i.e.*, searching for a pattern corresponding to some of the most significant bits of a concept or property identifier. Furthermore, allowing high rate compression and not requiring extra specific indexing process.

Finally, our solution focuses on query processing of SPARQL queries. It aims to minimize the memory footprint required during query execution and to perform optimizations in terms of SDS operations complexities: access, rank, rank_prefix, select and select_prefix. Adapting some of the heuristics presented in [20], the system optimizes execution of SDS operations. The ordering of basic graph patterns execution also takes into account simple statistics computed when generating the dictionaries (see Section 4.3).

[8] http://www.openrdf.org/

4 System description

We now describe the main components (Figure 1) of the current version of WaterFowl. Our system is composed of a dictionary based on automata and tries (already presented in Section 2.2), a two-layered datastructure and a query processing module. As mentioned, the dictionary only encodes subjects and non-concept objects entries of the dataset. Ontology elements are stored in the two-layered datastructure encoded in a clever way (compared to random and naive encodings) independent of the dictionary.

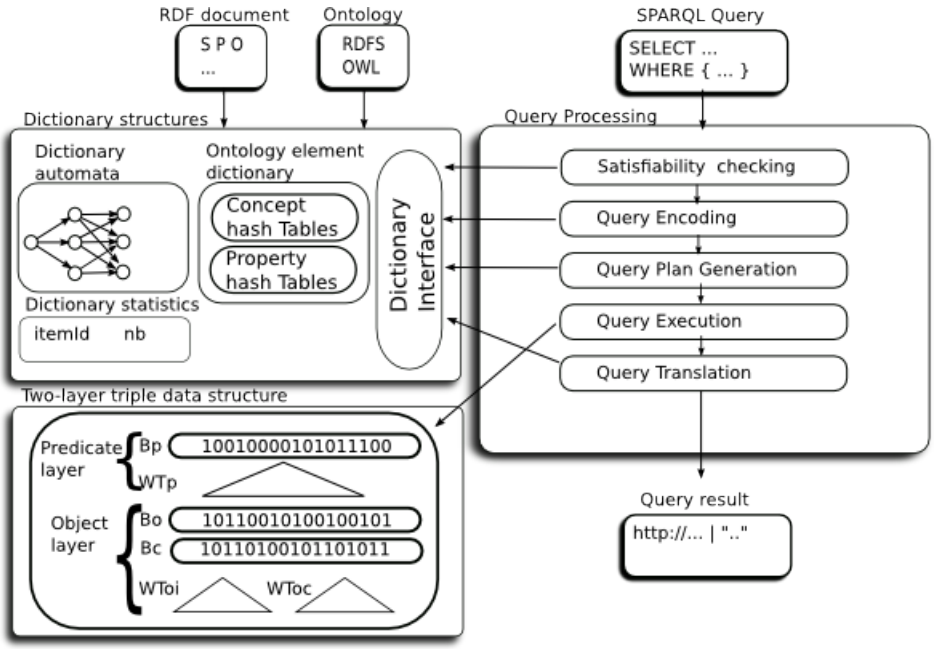

Fig. 1. WaterFowl's architecture

4.1 Two-layer structure

Once the dictionaries have been defined[9], the triples can be encoded (for readability reasons, in Figure 2(a) we represent identifiers with letters rather than long bit sequences) in a structure that makes an intensive use of SDS. To do so, the triples are ordered by subjects, predicates and then objects (see Figure 2(b)). The corresponding ordered forest is stored in a two-layer structure where each layer is composed of bitmaps and wavelet trees. The first layer encodes the

[9] encoding of ontology elements will be presented in next section

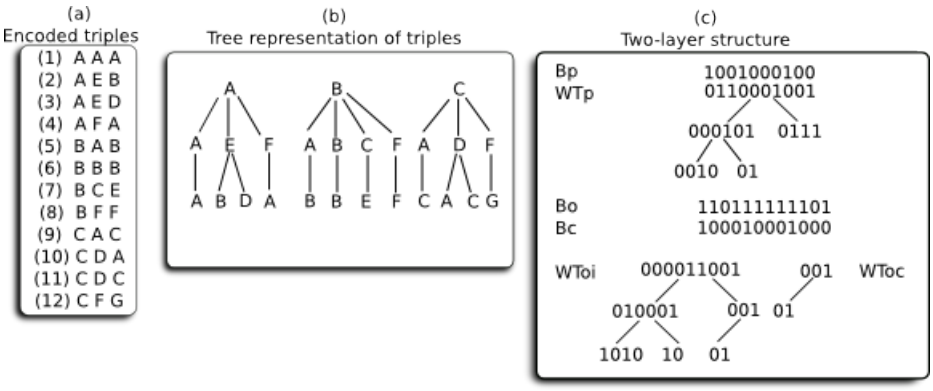

Fig. 2. Two-layer structure

relation between the subjects and the predicates; that is the edges between the root of each tree and its children.

The bitmap B_p is defined as follows. For each root of the trees in (Figure 2(b)) – that is each subject – the leftmost child is encoded as a 1, and the others as a 0. On the whole, B_p contains as many 1's as subjects in the dataset and is of length equal to the number of predicates in the dataset. In Figure 2(b), one obtains 1001000100 since there are 3 subjects with the first and third subjects having 3 predicates and the second one having 4. The wavelet tree WT_p encodes the sequence of predicates obtained from a pre-order traversal in the forest (*e.g.*, second row in Figure 2(b), $AEFABCFADF$). The construction of the wavelet tree follows the algorithm presented in Section 2.3.

Unlike the first layer, the second one has two bitmaps and two wavelet trees. B_o encodes the relation between the predicates and the objects; that is the edges between the leaves and their parents in the tree representation. Whereas, the bitmap B_c encodes the positions of ontology concepts in the sequence of objects obtained from a pre-order traversal in the forest (*e.g.*, third row in Figure 2(b), $ABDABBEFCACG$).

The bitmap B_o is defined as B_p considering the forest obtained by removing the first layer of the tree representation (that is the subjects). In Figure 2(b), one obtains 110111111101. The bitmap B_c stores a 1 at each position of an object which is a concept; a 0 otherwise. This is processed using a predicate contextualization, *i.e.*, in the dataset whenever a `rdf:type` appears, we know that the object corresponds to an ontology concept. In Figure 2(b), considering that the predicate `rdf:type` is encoding by 'A', one obtains 100010001000. Finally, the sequence of objects obtained from a pre-order traversal in the forest (*e.g.*, in Figure 2(b), $ABDABBEFCACG$) is splitted into two disjoint subsequences; one for the concepts and one for the rest (*e.g.*, in Figure 2(b), ABC and $BDABEFACG$ respectively). Each of these sequences is encoded in a wavelet tree (WT_{oc} and WT_{oi} respectively). This architecture reduces sparsity of iden-

tifiers and enables the management of very large datasets and ontologies while allowing time and space efficiency.

4.2 Ontology elements dictionary

Dictionaries presented in this section solely concern ontology elements, *i.e.*, concepts and properties (one for each). The aim of these last is to reply to the following expectations: (i) enable the transformation of SPARQL queries containing URIs and literals to queries consisting of integer identifiers, (ii) allow the transformation of integer-encoded results obtained from the query processing component into URIs and literals and (iii) support various inference-related operations such as a form of query transformation and satisfiability checking. Note that objectives (i) and (ii) are shared with our automata-based dictionary approach.

The ontology encoding is characterized by integer identifiers attributed to each ontology element entry. These integer values are possibly shared with entries of our other dictionaries but this is not an issue since we contextualize them. The generation of the ontology element dictionaries is performed off-line and we currently do not consider its ontology updates. That is, we know that each value appearing in the second position of a triple or of a SPARQL graph pattern is necessarily a property. Similarly for concept identifiers, we know that in the dataset their appearances as an object are associated with an `rdf:type` property. This identifier sharing characteristic among our different dictionaries opens up the encoding of very large set of identifiers, *e.g.*, 2^{32}-1 or 2^{64}-1 depending on the size of virtual memory addresses of the machine. We will see that the distribution of identifiers generated for the ontology dictionaries is qualified by a possibly high sparsity. Hence, enabling an encoding over a very large sets of identifiers ensures to support very large datasets and ontologies.

Our encoding methodology is directly motivated by the structure we are using in our two-layered datastructure. The overall aim is to encode the data itself and the ontology hierachies (that is the subsumption relation) in a compact way. To do so, the encoding will include in its definition the information of subsumption.

Prior to encoding, we are using a Description Logic reasoner to perform concepts classification, *e.g.*, Pellet or HermiT. Once the hierarchy is obtained, starting from the `owl:Thing` and an empty prefix, we recursively process each subconcept hierarchy of a concept C. That is, we compute the number of direct subconcepts of C and encode each of them on the minimum number of bits necessary[10] excluding the value 0. Indeed, value 0 is reserved to a special entry denoted `self`. This is motivated by the need of distinguishing a given concept from its set of subconcepts. Hence the encoding associated to `self` corresponds to a given concept (as it was a subconept of itself) while the identifier of the concept corresponds to its set of subconcepts (*e.g.*, 00 encodes any kind of Organization, whereas 00000 encodes specifically Organization excluding its subconcepts). This `self` mechanism is not required for the upper most hierarchy layer

[10] Note that one will try to fairly split the coding at each level in order to end up with a balanced-like wavelet tree inducing better operation complexities.

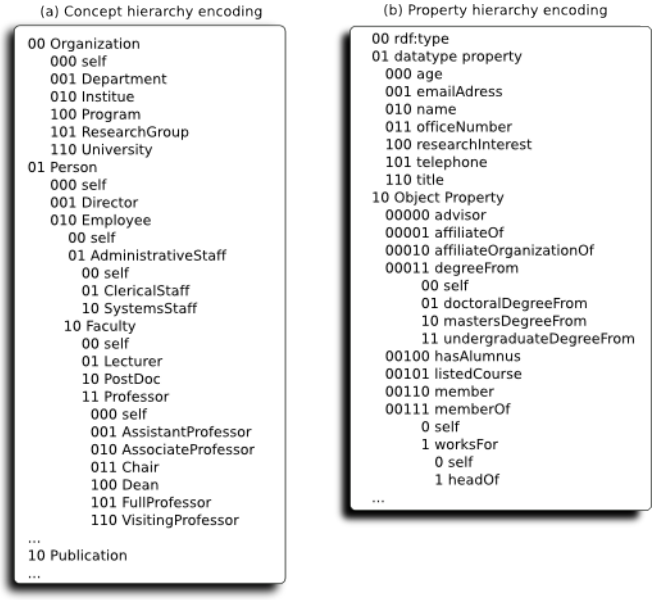

Fig. 3. Encoding for an extract of LUBM's ontology hierarchies

(*i.e.*, `owl:Thing`) since it is handled natively within our framework. Provided with this encoding one can easily query any entry regarding a given concept and its subconcepts by the use of rank_prefix and select_prefix operations.

Figure 3(a) represents an extract of LUBM's ontology. It emphasizes that the direct subsumption hierarchy of `owl:Thing` is encoded on 2 bits and that `Organization`, `Person` and `Publication` are respectively encoded at this level with values 00, 01 and 10. Moreover, `Organization` has 5 direct subconcepts and hence requires an encoding over 3 bits. At this level, `self` denotes the `Organization` concept (identified by the bit pattern 00 000 that is 00 as a prefix and 000 as its local identifier). Similarly, `Department` has a 00 prefix as well followed by a local encoding of 001, *i.e.*, a pattern 00 001.

Considering properties hierarchy, a reasoner is not necessary since no property classifications are required for the OWL ontologies we are dealing with. We also distinguish between the `rdf:type`, datatype and object properties encountered in the datasets. Hence, the first layer of our encoding consists of these three kinds of properties and are encoded on 2 bits. Otherwise, the approach is similar to the one defined for the concepts and also requires the `self` symbol to denote a property hierarchy. Figure 3(b) displays the property encodings for an extract of the LUBM's ontology.

Each of these process produces two hash tables: (i) one with an identifier as key and URI as value, denoted H_1, and (ii) one with URI as key and a tuple consisting of (a) an identifier, (b) the number of bits required to encode the direct sub-elements of this element and (c) some additional parameters such

as number of occurences, range and domain informations, denoted H_2. This additional informations are necessary to allow for the completeness of the RDFS entailment regime and to detect unsatisfiable queries, *e.g.*, when a SPARQL variable is bound to a concept C that is not instantiated in the dataset, which may require inferences, *i.e.*, rewriting the query such that the variable ranges over the subconcepts of C. It is also useful for reordering graph patterns to minimize the memory footprint of the executed query. For example, considering datasets generated from the LUBM, there is no instance for the `Professor` concept and LUBM's query #4 is unsatisfiable. Nevertheless, this query returns some results if the system seeks for all subconcepts of `Professor`. Note that our approach supports multiple inheritance by providing different identifiers to the same ontology element URI.

4.3 Query processing

The query processing component contains the modules displayed on the right part of Figure 1 and which coincides with the classical modules found in standard relational database management systems. Nevertheless, these modules are adapted to optimize performances of query answering in the context of an RDF data model and SDS operations. Due to space limitations, this section details the aspects related to query processing involving inferences and only provides general information on the aspects not requiring any form of reasoning, *i.e.*, we do not provide a complete presentation of our query optimization strategy which will be detailed in another paper. In the remaining of this section, we will ilustrate several aspects in the context of the LUBM [8] ontology with the following SPARQL query (henceforth denoted $QR1$) which seeks for pairs of `Professor/Department` satisfying the fact that the `Professor` works for that `Department`:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?x ?y WHERE {?y rdf:type lub:Department. ?x rdf:type lub:Professor.
?x lub:worksFor ?y.}
```

A first step consists in the parsing of a SPARQL query and checking for its syntactical validity. For each valid query, a satisfiability checking step is performed. It requires to communicate with the dictionary component to make sure that each element of a SPARQL graph pattern is present in the dictionaries. This is performed with both the automata based dictionary and the ontology element dictionaries through the use of a dictionary interface (Figure 1) which receives a set of basic graph patterns. Given a triple context (*e.g.*, search the object in the concept dictionary if the predicate is `rdf:type`), the system seeks in the right dictionary. The system detects two cases of unsatisfiablity: (i) one of the graph pattern's element (excluding variables) is not present in one of the dictionaries, (ii) a graph pattern element has no occurences in the datasets and, in the case of a concept or property has no sub-elements occurences neither. Otherwise, the query is satisfiable and the module obtains identifiers and statistics associated to each non variable graph pattern element. Note that in the case of a concept

or property element with sub-elements, it is the identifier associated to its `self` counterpart that is returned. In the case of $QR1$, the identifier and statistic associated to `Professor` are respectively '010101011000', *i.e.*, `Professor`'s `self` entry, and 0 since LUBM's datasets do not instantiate directly this concept. This approach enables to detect unsatisfiable queries rapidly since it detects that the query's result set is empty without executing any other steps of the query processing component.

A satisfiable query is then encoded in terms of identifiers retrieved from the set of dictionaries. It results in a query containing integer-based graph patterns and variables. In this step, the statistics associated to concept and property ontology elements encountered in graph patterns of the query may imply some form of reasoning. For instance, consider that such a concept C or property P has no instances, then since the query is satisfiable, it means that C or P has some sub-elements. Hence, some of its direct or indirect sub-elements may be instantiated and are expected in the result set of the query. The solution we are proposing is to replace the identifier of C or P's `self` entry with C or P's own identifier, *i.e.*, removing `self`'s local identifier in the query. In the context of $QR1$, it implies removing '000', `self`'s local identifier, from '010101011000' which yields to '010101011'. It corresponds to the `Professor` concept and is a common prefix to all its subconcepts. This encoding is strongly linked to the notion of SDS prefixed operations and will only requires to partially navigate in the associated wavelet tree. To do so our encoding supposes that the leftmost bit of the identifer is the most significant bit of the representation. In order to rewrite the query in terms of identifiers with a non constant number of significative bits, we adopt a strategy that add a 1 bit in front of our identifier. Considering our '010101011' identifier, we would then get '1010101011' (*i.e.*, 683) where the first one bit denotes the boundary of significant bits placed at its right. Clearly, this trick allows us to provide both the prefix of interest and the number of significant bits. Since we have not opted for inferences materialization, our database instances are not complete, *i.e.*, they do not contain all implicit information. In the case of the RDFS entailment regime, we can bypass this issue by using the `rdfs:domain` and `rdfs:range` properties to our advantage. For related properties, they enable to infer types for associated individuals. Let us demonstrate this aspect with an example. In the LUBM ontology, the axioms $\top \sqsubseteq \forall advisor^{-}.Person$ and $\top \sqsubseteq \forall advisor.Professor$ respectively defining that the `advisor` property has the concept `Person` as domain and `Professor` as range. Now consider a dataset where we have the following triples:

```
ex:smith ex:advisor ex:gblin.
ex:gblin lub:worksFor ex:esipe.
```

Moreover, we consider that no other triple explicitly types `gblin` as a `Professor`. Then query $QR1$ would not return `ex:gblin ex:esipe` in its answer set.

Our approach is supported by transforming the original query into a union query. That is, for each concept C involved in a graph pattern, we seek for properties that have C's identifier as a domain or a range. This information is available in the value of H_2 hashtable of the concept dictionary and can thus

be obtained efficiently. For each such property, we check, using the dictionary interface whether it is instantiated in the dataset. If it is not the case, we do not consider adding graph patterns for this property otherwise we add one as follows. In the case of a property P where C is the domain (resp. range), we rewrite the original query pattern by (i) removing the pattern `?x rdf:type C` involving C and (ii) replacing it with a pattern `?z P ?x` (resp. `?x P ?z`) where `?z` is a newly introduced variable which does not appear in the whole query graph patterns. This group of patterns is then unioned to the original query.

Applied to our previous example, that is retrieving the `ex:gblin ex:esipe` tuple, the system will rewrite $QR1$ into
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?x ?y WHERE {{?y rdf:type lub:Department. ?x rdf:type lub:Professor. ?x lub:worksFor ?y.} UNION {?y rdf:type lub:Department. ?x lub:worksFor ?y. ?z lub:advisor ?x. }

Note that in the case where the `Professor` concept has no instances in a dataset, the first set of graph patterns unioned in the query is not necessary since it will return an empty answer set.

A best effort query plan is then searched using a set of heuristics. A first one is especially designed to reduce the cost of navigating in the two-layer structure, in terms of rank, select and access SDS operations. That is we try as much as possible to favor rank operations against select ones since most implementations guarantee constant time rank operations on bitmap but not for select ones which either need lot of extra space or logarithmic time. Two other heuristics are provided to take advantage of state of the art RDF access pattern [20], [19] and statistics stored in the dictionary structures. Again, these heuristics have been adapted to reorder some access patterns which is a major source of optimizations for SPARQL queries containing many graph patterns. This results in the generation of query plans taking the form of left-deep join trees which is being translated and executed in terms of compositions of rank, select and access SDS operations. In order to support `distinct`, `limit`, `offset` and `order by` SPARQL operators, we provide a k-partite graph based storing system for the candidate tuples that allow us to store and filter them in an efficient way avoiding as much as possible unneccesary cartesian product. Finally, the identifiers of the result are translated in terms of their associated value in the dictionaries.

The supported SPARQL operators needed the development of optimization techniques in the query execution module: the `union` of graph patterns which is based on a lazy approach of common patterns, `filter` which requires accesses to the dictionary and `optional` that prevents the creation of bindings in the absence of a matching for the optional graph patterns.

5 Experimental evaluation

5.1 System

All experiments have been conducted on a HP Z800 workstation with 2 Quad-Core Intel Xeon Processors with 12Mbytes L2 cache, 24Gbytes of memory and running Gentoo 2.6.37 generic x86-64. It contains two 500GB SATA disks running at 7200 rpm. We used gcc version 4.5.2 running on 64 bits with glibc 2.13. We modified the libcds v1.0.13 in order to obtain rank_prefix and select_prefix operations on the proposed SDS. We have compared our system with RDF-3X version 0.3.7. We do not propose a comparison with Hexastore since it was not possible to load the datasets we are working with. This is due to its in-memory approach and the large number of set indexes, *i.e.*, 6, it requires to process queries efficiently. Note that this aspect was confirmed in [12] which essentially focusing on data loading, compression rates and times required for indexes creation. Our current WaterFowl framework uses pointer-free wavelet trees (which were giving best results compared to pointer based wavelet trees and wavelet matrices).

5.2 Datasets

In this section, we present the results of our evaluation performed on a set of synthetic and a real world datasets. The synthetic datasets correspond to instances of the Lehigh University Benchmark (LUBM) [8]. The main characteristics of LUBM are to feature an OWL ontology for the university domain, to enable scaling of datasets to an arbitrary size and to provide a set of 14 SPARQL queries of varying complexities. Out of these queries, 10 require a form of inference, namely dealing with concept and property hierarchies as well as inverse and transitive roles which we are not testing since they require OWL entailments. We are testing our system on two datasets, one for 100 and another one for 1000 universities. Table 1 summarizes the sizes in space and number of RDF triples of all datasets. The real world datasets corresponds to Yago and is mainly used on the first aspect of our evaluation.

Table 1. Description of the datasets

Dataset	Triples (Million)	Size (MB)
LUBM100	13.4	1125
LUBM1000	133.5	11307
Yago2	37.5	5325

5.3 Results

The results we are presenting in this section concern three aspects of our system: (i) memory footprint and time required to prepare a dataset, (ii) query processing

Table 2. Size of database serialization (MB) and Time to prepare datasets

	Size in MB			Time in sec		
	univ100	univ1000	Yago	univ100	univ1000	Yago
RDF3X	831,717	7,795,458	2,189,735	240	3050	1090
WaterFowl Mode 1	73,231	737,685	217,293	168	2134	768
WaterFowl Mode 2	56,851	576,317	168,445	119	1515	545
WaterFowl Mode 3	61,881	639,063	162,982	107	1488	513

not requiring inferences and (iii) query answering requiring RDFS entailment regime.

The first one aims to demonstrate that a system designed on SDS possesses interesting properties in terms of data compression rate, time to prepare a dataset, *i.e.*, total duration required to create the dictionary, index the data, compute some statistics and serialize the database structure. It is presented in Table 2 and confirms the results contained in [12]. We can see that most compressed versions of WaterFowl, mode 2 and 3 relying respectively on non-pointer and so-called matrix wavelet trees require between 5 and 9% of the space required by RDF3X. This is due to the high compression rate of the SDS we are using and the single, opposed to 15, index we are generating. Moreover, times to prepare a dataset are about half of the duration taken by RDF3X. This is easily explained by the number of indexes RDF3X is building. Finally, our Mode 2, based on a wavelet tree pointer-free implementation seems to be a interesting tradeoff between size of the generated dataset and generation time.

The two next aspects of our evaluation concerns query processing. First we consider queries that are not requiring reasoning then we study some queries requiring the RDFS entailement regime. We consider that by investigating both aspects of query answering, we are able to highlight that the pros and cons of our complete query processing component.

Table 3. Query answering times (sec) on univ1000

	LUBM QR#1	LUBM QR#2	LUBM QR#14
RDF3X	1.65	14.88	1640
WaterFowl Mode 2	1.80	10.18	1710
WaterFowl Mode 3	1.75	10.13	1680

In the first context, we compare our approach with RDF3X on a subset of LUBM queries (#1, #2 and #14). Table 3 emphasizes that the performances of both system are comparable. Note that these queries have different characteristics since they respectively correspond to large input with high selectivity, complex 'triangle' query pattern and large input with low selectivity.

In the context of queries requiring RDFS entailement, we are testing RDF3X with query rewriting performed using a DL reasoner against our system. That is,

Table 4. Inference-based query answering times (sec) on univ100

	QR#4	QR#5	QR#6	QR#7	QR#10
RDF3X	4.2	2.5	15.3	1.4	1.6
WaterFowl Mode 2	2.66	1.3	13.1	1.2	1.4

we have implemented a simple RDFS query rewriting on top of RDF3X which generates SPARQL queries with `union` clauses. The RDF3X approach enables to perform query rewriting in the context of the considered fastest RDF Store. Table 4 highlights that our system slightly outperforms the inference-enable RDF3X on a set of five distinct LUBM queries, requiring different forms of reasoning, *i.e.*, based on concept and property subsumption relationships.

6 Conclusion

We have designed and implemented a novel type of RDF store that addresses a set of issues of big data and of the semantic web. Each database instance regroups a set of dictionaries and a dataset represented in a compact, self-indexed manner using some succinct data structures. The evaluation we have conducted emphasize that our system is clearly very efficient in terms of data compression and can thus be considered as an interesting alternative when one is concerned with data exchange. Moreover, on our query processing experimentations, our system presents performances that are comparable to the domain's reference, *i.e.*, RDF3X. We consider that this quite a strong encouragement toward pursuing our work on WaterFowl. We consider that this is due to the advantage of our highly compressing the data and especially implementing all data retrieving operations on SDS functions, *i.e.*, access, rank, prefix and some prefic counterparts. We also believe that founding and adapting all our query optimization heuristics on state of the art solutions is part of the good performances our system provides. Nevertheless, we are convinced that there is plenty of room for more optimizations in all modules of WaterFowl, *e.g.*, pipelined parallelism in query execution.

Our future experiments with WaterFowl suggest a promising direction for future investigations. They will mainly include the distribution of triples over a cluster of machines and the support for updates in both the TBox and the ABox of the knowledge base. Considering the latter, we are investigating incremental solutions while an amortized approach is considered on the issue of updating the ABox. Finally, we would like to propose extensions of our ontology dictionary that go beyond the RDFS entailment regime, e.g. OWL2 entailement.

References

1. D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

2. M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. In *WWW*, pages 41–50, 2010.
3. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International Semantic Web Conference*, pages 54–68, 2002.
4. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
5. J. D. Fernández, M. A. Martínez-Prieto, and C. Gutierrez. Compact representation of large rdf data sets for publishing and exchange. In *International Semantic Web Conference (1)*, pages 193–208, 2010.
6. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850, 2003.
7. R. Grossi and G. Ottaviano. The wavelet trie: maintaining an indexed sequence of strings in compressed space. In *PODS*, pages 203–214, 2012.
8. Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
9. S. Harris and A. Seaborne. SPARQL 1.1 query language W3C recommendation. http://www.w3.org/tr/sparql11-query/, 2013.
10. P. Hayes. RDF semantics, W3C recommendation. http://www.w3.org/tr/rdf-mt/, 2004.
11. G. Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554, 1989.
12. M. A. Martínez-Prieto, M. A. Gallego, and J. D. Fernández. Exchange and consumption of huge rdf data. In *ESWC*, pages 437–452, 2012.
13. J. I. Munro. Tables. In *FSTTCS*, pages 37–42, 1996.
14. T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *VLDB J.*, 19(1):91–113, 2010.
15. H. Pérez-Urbina, I. Horrocks, and B. Motik. Efficient query answering for owl 2. In *International Semantic Web Conference*, pages 489–504, 2009.
16. D. Revuz. *Dictionnaires et lexiques : mthodes et algorithmes*. PhD thesis, Paris 7, 1991.
17. M. Rodriguez-Muro and D. Calvanese. High performance query answering over dl-lite ontologies. In *KR*, 2012.
18. R. Rosati and A. Almatelli. Improving query answering over dl-lite ontologies. In *KR*, 2010.
19. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *WWW*, pages 595–604, 2008.
20. P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. A. Boncz. Heuristics-based query optimisation for sparql. In *EDBT*, pages 324–335, 2012.
21. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.