WaterFowl: A Compact, Self-indexed and Inference-Enabled Immutable RDF Store

Olivier Curé[1], Guillaume Blin[1], Dominique Revuz[1], and David Célestin 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 WaterFowl, a novel approach for the storage of RDF triples that addresses scalability issues through compression. The architecture of our prototype, largely based on the use of succinct data structures, enables the representation of triples in a self-indexed, compact manner without requiring decompression at query answering time. Moreover, it is adapted to efficiently support RDF and RDFS entailment regimes thanks to an optimized encoding of ontology concepts and properties that does not require a complete inference materialization or query reformulation. This approach implies to make a distinction between the terminological and the assertional components of the knowledge base early in the process of data preparation, *i.e.*, preprocessing the data before storing it in our structures. The paper describes our system's architecture and presents some preliminary results obtained from evaluations on different datasets.

Keywords: #eswc2014Cure.

1 Introduction

The emergence of big data imposes 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 data volumes focus on parallel processing, *e.g.*, with MapReduce [3] like 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.

Due to the production of an increasing number of voluminous datasets, RDF (Resource Description Framework) is concerned with this phenomenon. In this data model, a triple is made up of a subject, a property and an object and is generally represented as a graph. To foster interoperability among applications manipulating RDF data, vocabularies such as RDFS (RDF Schema) and OWL (Web Ontology Language) have been defined in the context of the W3C's Semantic Web Activity. They support further means to describe the structure and semantics of RDF graphs and are themselves expressed as RDF triples. When considered together, RDF data and its vocabulary represent a knowledge base

V. Presutti et al. (Eds.): ESWC 2014, LNCS 8465, pp. 302–316, 2014.

which presents the main advantage of consistently managing the data and metadata within the same data model. In the context of a Semantic Web knowledge base, handling inferences adds to the list of standard query processing database management issues, *i.e.*, parsing, optimizing and executing a query.

In this paper, we design a new architecture for immutable RDF database systems that addresses compression and inference-enabled query answering and evaluate it using a proof of concept prototype (see Section 5). This framework will serve as the cornerstone for upcoming features that will include data partitioning and supporting data updates and thus becoming mutable. The foundation of our system consists of a high compression, self-indexed storage structure supporting 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 – see Section 2) provide such properties and are extensively used in our architecture (especially wavelet trees). The high rate compression obtained from SDS enables the system to keep a large portion of the data in-memory. Moreover, efficient SDS serialization/deserialization operations support fast disk-oriented IOs, *e.g.*, data loading. Based on a preliminary work of Fernández *et al.* called HDT (Header Dictionary Triples) [4] – 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 (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 in [4] nor in [9]. The basic idea is to use an encoding of the data that will capture the subsumption relationships of both concepts and properties. Therefore, the encoded data will enclose – without extra cost – both raw data and ontology hierarchies. Our approach differs from existing ones, such as in [13], since our encoding, which is prefix-based, will allow us to further restrict the number of SDS operations needed to answer a query implying inference. To do so, the system will need to adapt standard *rank* and *select* wavelet tree operations into ones that consider prefix of binary encoded identifiers [7]. This solution will spare the use of an expensive query rewriting approach or complete inference materialization (via a forward-chaining approach) when requesting a given ontology element, *i.e.*, concept or property, and all its sub-elements. In order to complete RDFS entailment regime, we address *rdfs:domain* and *rdfs:range* through a minimalist materialization of subject, respectively object, *rdf:type* properties.

2 Succinct Data Structures

The family of SDS uses a compression rate close to theoretical optimum, but simultaneously allows efficient decompression-free query operations on the compressed data. This property is obtained using a small amount of extra bits to store additional information. Bit vectors (*a.k.a.* bit maps) 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 [10]

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 $access(x)$). In the remainder of this paper, we do not precise the bit b anymore for these operations and simply write *rank* and *select*. 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 split into two equal parts. One attributes a 0 to each character of the first part and a 1 to the other. 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 – referred as *rank_prefix* and *select_prefix* operations (that will be of great interest for us along with our encoding of ontology concepts and properties). Wavelet trees have been well studied since then and both space and time efficient implementations are now available, *e.g.*, the libcds library[1] which we have extended with rank and select prefix operations.

3 Related Work

Abadi *et al*'s paper [1] reinvigorated the development of novel approaches to design RDF engines. In particular, performance of query processing started to get more attention. Solutions such as RDF-3X [11] were designed using multiple indexes to address this issue but index proliferation also came at the cost of high memory footprint. Matrix Bit loaded [2] is another multiple indexes solution which stores compressed data into bit matrices. Comparatively, our approach proposes a single structure that enables indexed access on all triple elements.

Our approach is inspired by the HDT [4] system which mainly focuses on data exchange (and thus on data compression). Its former motivation was to support the exchange of large datasets highly compressed using SDS. Later, [9] presented HDT FoQ, an extension of the structure of HDT that enables some simple data retrieving operations. Nevertheless, this last contribution was not allowing any form of reasoning nor was detailing query processing mechanisms. In fact, WaterFowl brings the HDT FoQ approach further to its logical conclusion by using a pair of wavelet trees in the object layer (HDT FoQ uses an adjacency list for this layer) and by integrating a complete query processing solution with complete RDFS reasoning (*i.e.*, handling any inference using RDFS

[1] `https://code.google.com/p/libcds/`

expressiveness). This is made possible by an adaptation of both the dictionary and the triple structures. Note that this adaptation enables to retain the nice compression properties of HDT FoQ (see Section 5).

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 reformulation at run-time but is associated with an expansion of the memory footprint and difficulties to handle update operations. To address this last issue, [5] proposes to qualify each triple with a boolean value that states whether a triple is the result of a previous inference and a count on the triple appearance in the data set. One advantage of this approach is to consider updates at both the ABox and TBox levels but it requires a larger memory footprint. Another approach consists in performing query rewriting at run-time. It guarantees a light memory footprint but it is associated with a significant increase of queries generated. Presto [14] and Requiem [12] are systems adopting this approach with different algorithms. By adopting a rewriting approach into non recursive datalog, Presto achieves to perform this operation in non exponential time. The encoding of ontology elements, *i.e.*, concepts and properties, used in our system is related to a third approach which consists in encoding elements in a clever way that retains the subsumption hierarchy. This is the approach presented in [13] and implemented in the Quest system (a relational database management system). The work of Rodriguez-Muro *et al.* [13] 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 tackles the encoding at the bit level and focuses on the sharing of common prefixes in the encoding of the identifiers. The main benefit of this approach compared to [13] is that the queries may be rewritten in terms of *rank_prefix* and *select_prefix* operations which will not require a full deep traversal of the wavelet trees (*i.e.*, inducing less operations on the SDS). Furthermore, it allows high rate compression and does not require extra specific indexing processes. 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*. Moreover, our query optimizer also takes into account triple pattern heuristics adapted from [15] as well as some simple statistics computed when generating the dictionaries.

4 System Components

4.1 Dictionary Component

In WaterFowl, dictionaries (see Figure 1) are used to: (i) transform the triple patterns, called Basic Graph Pattern (BGP), of SPARQL queries into their encoded version, (ii) transform the encoded result of a query back to their original verbose values and (iii) support various inference-related operations such as a

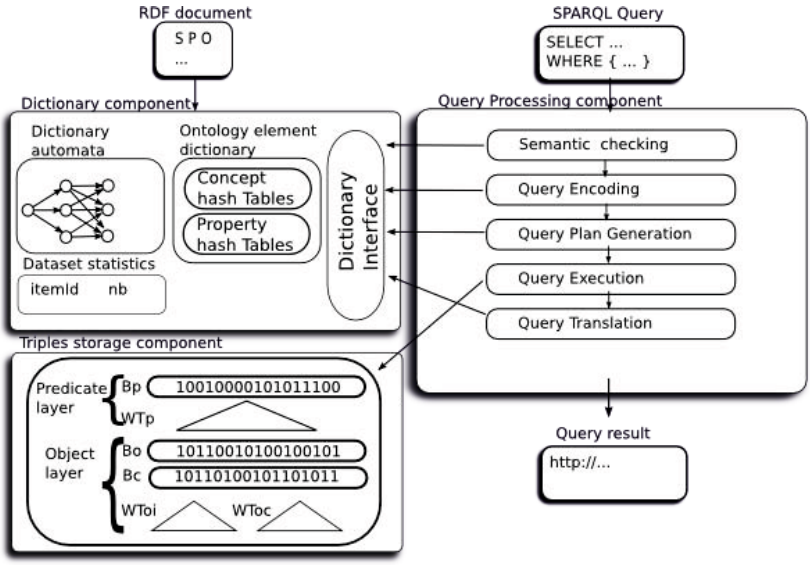

Fig. 1. WaterFowl's architecture

form of query transformation and semantic checking. Our dictionary structures are responsible for storing some simple data statistics. They are used for query optimization and are much simpler than histograms found in RDBMS due to the prohibitive size and time that would be required to respectively store and compute them. The stored statistics correspond to the total number of subjects, predicates and objects in the dataset as well as the number of occurrences of distinct subjects, predicates and objects. These statistics mainly help in discovering the most cost-efficient physical plan of a given query. We will provide more details in Section 4.3. The dictionary interface supports the communication between the query processing and the dictionary components.

In the remainder of this section, we focus on the ontology dictionaries, *i.e.*, concepts and properties (one for each), which is performed off-line. Details on the instance dictionary, which is based on common dictionary practices, are omitted due to space limitation. 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, *i.e.*, an integer can identify both an instance, a property and a concept, without ambiguities since they are contextualized. That is, we know that each value appearing in the second position of a triple or of a SPARQL triple 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. Since our method to handle SPARQL BGP is based on navigating through our two-layered structure, we always get the information required to consider the context. This identifier sharing char-

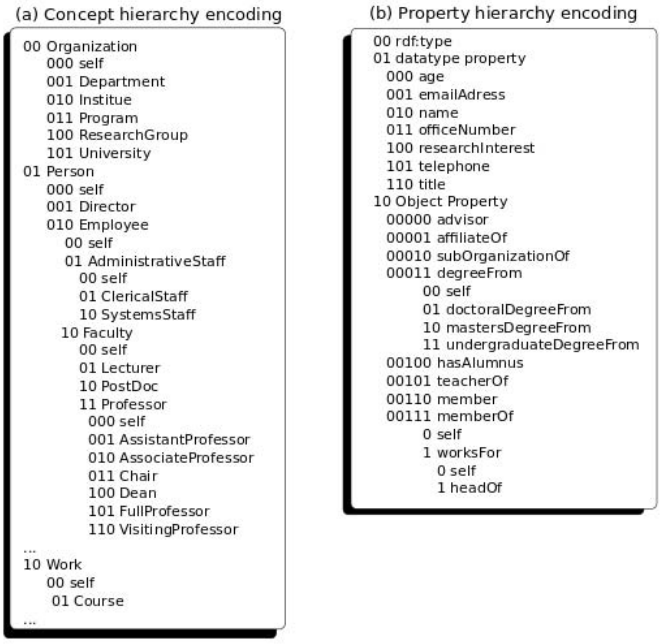

Fig. 2. Encoding for an extract of LUBM's ontology hierarchies

acteristic among our different dictionaries opens up the encoding of large set of identifiers, regardless of the structure of concept and property hierarchies. 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 large sets of identifiers ensures to support large datasets and ontologies. The overall objective is to encode the data itself and the ontology hierarchies (that is the subsumption relations) in a compact way.

Prior to encoding, we are using a Description Logic reasoner, *e.g.*, Pellet[2], to perform the classification of concepts. Note that this approach enables to consider ontologies more expressive than RDFS, *e.g.*, OWL2DL. Then, we navigate in a breadth-first search manner through this classification. This enables to compute the representation of all concepts such that any pair of concepts sharing a common ancestor in the concept hierarchy will share a common prefix in their representation (corresponding to this common ancestor). To do so, starting from the *owl:Thing* and an empty prefix, we compute the number of direct subconcepts of *owl:Thing*. We encode each of these last with a minimum number of bits. The encoding of each such concept will be a common prefix to the encoding of any concept belonging to its sub-hierarchy (based on the subsumption relation). Figure 2(a) represents an extract of the Lehigh University Benchmark (LUBM) ontology. It emphasizes that `owl:Thing`'s direct subsumption hierarchy

[2] `http://clarkparsia.com/pellet/`

is encoded on 2 bits and that any subconcept of *Organization* (resp. *Person*, *Work*) is encoded with prefix 00 (resp. 01, 10). We will now act in a similar way for each direct subconcepts of *owl:Thing*. The only difference being the assumption that any concept (except *owl:Thing*) has a direct subconcept named *self*. This is needed to differentiate, in query processing, a query targeting a given concept (referred as *self*) or its set of subconcepts. For ease of treatment, we will always attribute the 0 value to *self*. Hence the encoding associated to *self* will correspond to a given concept (as if it was a subconcept of itself) while the identifier of the concept corresponds to its set of subconcepts. For example, querying any concept encoded with the prefix 00 will correspond to seeking for any kind of *Organization* while querying any concept encoded with the prefix 00 000 will seek specifically for *Organization* excluding its subconcepts. Indeed, the prefix 00 000 excludes *Department* which is encoded by the prefix 00 001 while the prefix 00 includes all kind of *Organization*. Recall that we use $rank\_prefix$ and $select\_prefix$ operations which differentiate 00 and 00 000. By recursively processing the hierarchy of concepts, one will end up with a prefix encoding (as illustrated in Figure 2). This *self* mechanism is not required for *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. Considering the properties, we first distinguish between the *rdf:type*, datatype and object properties encountered in the datasets and assign specific prefixes of 2 bits to each of them (resp. 00, 01 and 10). For both the sets of object and datatype properties, we apply a similar process as for the concepts in order to achieve a prefix encoding. Figure 2(b) displays the property encodings for an extract of the LUBM's ontology.

The corresponding encodings are stored in two types of 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, (c) some additional parameters such as number of occurrences, finally (d) range and domain information, denoted H_2. This additional information 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.*, modifying the query such that the variable ranges over the subconcepts of C. It is also useful for reordering graph patterns in order 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*.

Our approach is adapted to tree-like hierarchies. Nevertheless, we can support multiple inheritance of ontology entities in several ways. First of all, in order to capture all the knowledge, one would have to use different prefixes for the same ontology entity. For example, let us consider a concept A having X and Y as super-concepts respectively identified by the prefixes 00 and 01. Our solution relies on providing a single prefix to any concept – even the ones with multiple

super-concepts. Arbitrarily, we decide to assign the prefix corresponding to the first super-concept encountered in the data. Hence, all occurrences of a concept in the dataset will share a single common prefix. There will be no expansion of the dataset. In order to be able to derive all the knowledge induced by the multiple inheritance, we introduce an `equivalence` data structure that provides all encodings for a given concept. Considering our previous example, concept A appears in the data as 01 10 as well as 00 01. Our solution, will thus use some query rewriting techniques to retrieve all information induced by the multiple inheritance. For example, if one wants to retrieve all information regarding any sub-concept of X, this request should require any concept encoded using the prefix 00. Queries requiring to access the content of the `equivalence` structure contain `UNION` clauses to address all its (sub-) concepts. Let the encoding of A be 00 01, in order to retrieve any information of Y or of one of its sub-concepts, one will ask for the union of any concept with prefix 01 or 00 01 since in the equivalence data structure, 01 01 is equivalent to 00 01. Even if this approach has some drawbacks (possibly heavy query rewriting), one only needs to efficiently know which subsumption relation are not directly expressed in the data and to store multiple inheritance for the direct common sub-concept only (which clearly are rare). On the whole, this solution seems more acceptable for our purpose than heavy materialization.

4.2 Triples Storage Component

Once the dictionaries have been defined, the triples can be encoded in a structure that makes intensive use of SDS. To illustrate the structure, we will encode the following simple RDF triples: {`(Uni0, rdf:type, ub:University), (Uni0, ub:name, "University0"), (Dpt0, rdf:type, ub:Department), (Dpt0, ub:name, "Department0"), (Dpt0, ub:subOrganizationOf, Uni0), (AP0, rdf:type, ub:AssociateProfessor), (AP0, ub:name, "Cure"), (AP0, ub:teacherOf, C15), (AP0, ub:teacherOf, C16), (AP0, ub:worksFor, Dpt0), (C15, rdf:type, ub:Course), (C15, ub:name, "Course15"), (C16, rdf:type, ub:Course)`} .

The triples are first ordered by subjects, predicates and then objects. The ordered forest of Figure 3(a) will serve to demonstrate the creation of our two-layered structure where each layer is composed of bitmaps and wavelet trees.

The first layer encodes the 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 3(a) – 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 3(c), one obtains 101001000101 since there are 5 subjects with the last subject having 1 predicate, the first and fourth subjects having 2 predicates, the second one having 3 while the third one has 4. The wavelet tree WT_p encodes the sequence of predicates obtained from a pre-order traversal in the forest (*i.e.*, second row in Figure 3(a)). The construction of the wavelet tree follows the method described in Section 2.

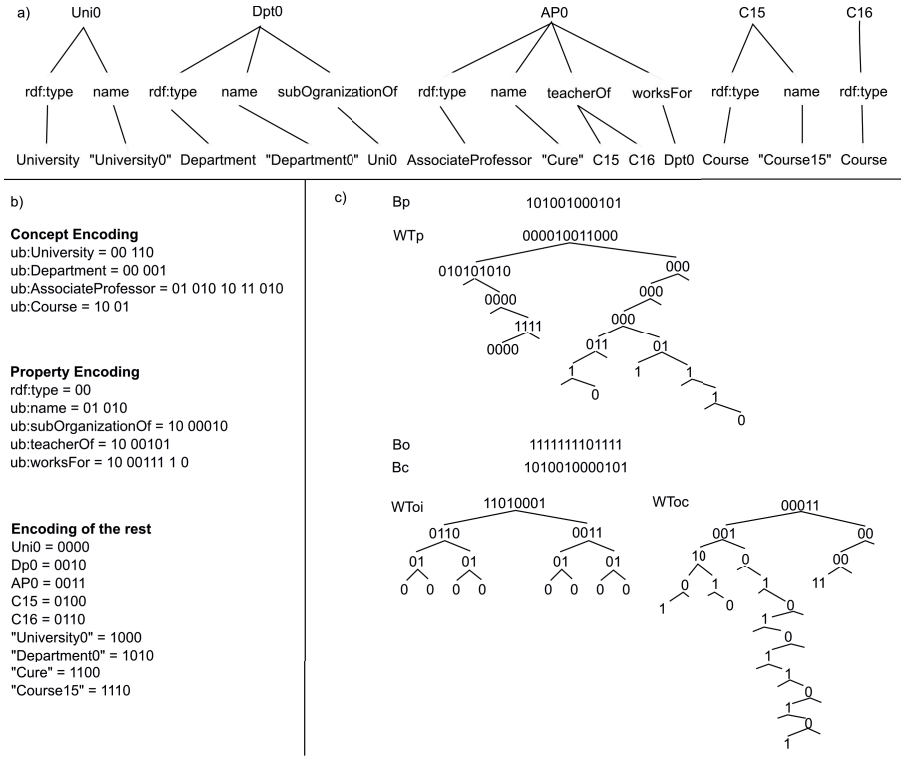

Fig. 3. Two-layered structure. For ease of presentation, URIs have been shorten. a) Tree-like representation of some RDF triples. b) Encodings. c) Corresponding storage.

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 (*i.e.*, third row in Figure 3(a)). 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 3(a), one obtains 1111111101111. 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 3(a), considering that the predicate *rdf:type* is encoded by 00, one obtains 1010010000101. Finally, the sequence of objects obtained from a pre-order traversal in the forest (*i.e.*, third row in Figure 3(a)) is split into two disjoint subsequences; one for the concepts and one for the rest. Each of these sequences is encoded in a wavelet tree (resp. WT_{oc} and WT_{oi}). This architecture reduces sparsity of identifiers and enables the management of very large datasets and ontologies while allowing time and space efficiency.

4.3 Query Processing Component

The query processing component contains the modules displayed on the right part of Figure 1. It 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. In the remainder of this section, we will illustrate 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*: `SELECT ?x ?y WHERE {?y rdf:type ub:Department. ?x rdf:type ub:Professor. ?x ub:worksFor ?y.}`

A first step consists in parsing the SPARQL query and checking for its well-formedness. For each valid query, a semantic checking step is performed. It first involves 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 instance and ontology dictionaries through the use of a dictionary interface (Figure 1) which receives a set of triples of the BGP. Given a triple context, the system seeks in the appropriate dictionary (*e.g.*, search the object in the concept dictionary if the predicate is *rdf:type*). The system detects two cases of unsatisfiability: (i) one of the graph pattern's element (excluding variables) is not present in any of the dictionaries, (ii) a graph pattern element has no occurrences in the datasets and, in the case of a concept or property, has no instantiated sub-elements occurrences neither. Otherwise, the BGP 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 01 010 10 11 000, *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 query is considered unsatisfiable if at least one triple of the BGP is unsatisfiable otherwise, the whole query is satisfiable.

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 concepts and properties encountered in the BGP may imply some form of reasoning. For instance, consider that 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 01 010 10 11 000 which yields to 01 010 10 11. It corresponds to the *Professor* concept and is a common prefix to all its subconcepts.

To complete the support for RDFS entailment regime, we have implemented a materialization-based approach for the *rdfs:domain* and *rdfs:range* properties. Intuitively, we provide, if one is not available, or refine, if one is available, a type to the subject (resp. object) of a triple whose property has some domain (resp. range) information. The refinement aspect corresponds to typing a subject (or object) with the most specific concept among a set of valid ones. This enables to limit the size of our materialization by preventing over typing. Our typing mechanism is based on the following RDFS entailment rules (denoted rdfs2 and rdfs3 in the RDF Semantics W3C Recommendation[3]): (i) if *aaa rdfs:domain xxx. uuu aaa yyy.* then add *uuu rdf:type xxx.* and (ii) if *aaa rdfs:range xxx . uuu aaa vvv.* then add *vvv rdf:type xxx.*

Let us demonstrate this aspect with an example. In the LUBM ontology, the axioms $\top \sqsubseteq \forall\ advisor^{-}.Person$ and $\top \sqsubseteq \forall\ advisor.Professor$ resp. define that the *advisor* property has the concept *Person* as domain and *Professor* as range. Let also consider a dataset with the {`(ex:smith, ub:advisor, ex:gblin)`, `(ex:gblin, ub:worksFor, ex:esipe)`} triples. We now present two cases where, according to the RDFS entailment regime, the `(ex:gblin, ex:esipe)` tuple should be part of the $QR1$ answer set. In a first case, we assume that neither `gblin` nor `smith` are typed. Then our materialization strategy adds the triples {`(ex:smith, rdf:type, Person)`, `(ex:gblin, rdf:type, Professor)`} and ensures the completeness of the answer set. In a second case, the triple {`(ex:gblin, rdf:type, Person)`} was part of the dataset and will be replaced by {`(ex:gblin, rdf:type, Professor)`} since `Professor` is more specific than `Person`. In cases of multiple incomparable classes for which no "most specific" class exists then several types are stored for the considered instance. These materializations are part of our data preprocessing and make an intensive use of the H_2 structure. In Section 4.1, we emphasized that the H_2 structure stores in its value the concepts associated to the *rdfs:domain* and *rdfs:range* of each property. Finally, note that this solution does not come at the cost of expanding our two-layered structure and it does not imply any query reformulation.

We now sketch the main aspects of our query execution and optimization process. A best effort query plan is searched using a set of heuristics. A first one is especially designed to reduce the cost of navigating in the two-layered 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 [15], 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

[3] `http://www.w3.org/TR/rdf-mt/#rdf_entail`

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 (and their prefix forms). 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 unnecessary Cartesian product. Finally, the identifiers of the result are translated in terms of their associated values in the dictionaries. The supported SPARQL operators needed the development of optimization techniques in the query execution module: the `UNION` of graph patterns requires a lazy approach of common patterns, `FILTER` accesses the dictionary and `OPTIONAL` prevents the creation of bindings in the absence of a matching for the optional graph patterns.

5 Experimental Evaluation

5.1 System and Datasets

All experiments have been conducted on a HP Z800 workstation with 2 Quad-Core Intel Xeon Processors with 12Mbytes L2 cache, 8Gbytes 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, BigOWLIM version 3.5 and Jena 2.6.4 together with its TDB 0.8.10. 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 [9] which essentially focuses 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).

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 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 datasets for 100 and 1000 universities, resp. 13.4 (1.12Gb) and 133.5 (11.3Gb) million triples The real-world dataset is Yago (37.5 million triples and 5.32Gb) and is mainly used on the first aspect of our evaluation.

5.2 Results

The results concern three aspects of our system: (i) memory footprint and time required to prepare a dataset, (ii) query processing 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 1 and confirms the results contained in [9]. We can see that most compressed versions of WaterFowl, *i.e.*,mode 2 and 3 relying respectively on pointer-free wavelet tree and wavelet matrix (while mode 1 uses pointers) require between 5 and 9% of the space required by RDF-3X and this is even more important compared to BigOWLIM and Jena TDB. This is due to the high compression rate of the SDS we are using and the single, opposed to 15 for RDF-3X, index we are generating. The sizes required for BigOWLIM and Jena TDB are explained by their approach which require full materialization. Moreover, times to prepare a dataset are about half of the duration taken by RDF-3X. This is easily explained by the number of indexes RDF-3X is building. Obviously, due to the materialization, the times needed to process and store the datasets are even more important for BigOWLIM and Jena TDB. Finally, our mode 2 (pointer-free Wavelet), based on a pointer-free wavelet tree implementation seems to be an interesting trade-off 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 entailment regime. We consider that by investigating both aspects of query answering, we are able to highlight the pros and cons of our complete query processing component. Our evaluation methodology includes a warm-up phase before measuring the execution time of the queries. This is required for the 3 compared systems but not for WaterFowl since its data reside in main-memory. All the queries are first ran in sequence once to warm-up the systems, and then the process is repeated 5 times. The following tables (Table 2) report the mean values for each query and each system.

In the first context, we compare our approach with the 3 other systems on a subset of LUBM queries (#1, #2 and #14). Table 2 emphasizes that the performances with the RDF-3X system are comparable. Unsurprisingly, the two other systems are slower than RDF-3X on Queries #1 and #3. A fact which has been highlighted on many other evaluations. Note that these queries have different characteristics since they respectively correspond to large input with

Table 1. Size of database serialization (MB) and Time to prepare datasets

	Size in MB			Time in sec		
	univ100	univ1000	Yago	univ100	univ1000	Yago
RDF-3X	831,717	7,795,458	2,189,735	240	3050	1090
BigOWLIM	2,411,260	22,600,088	6,348,338	838	10640	3708
Jena TDB	1,492,057	13,984,467	3,928,271	1285	16332	5837
WaterFowl Mode 1	91,539	922,106	271,616	168	2134	768
WaterFowl Mode 2	71,064	720,396	210,556	119	1515	545
WaterFowl Mode 3	77,351	798,829	203,728	107	1488	513

Table 2. Query answering times (sec) on univ1000

	LUBM QR#1	LUBM QR#2	LUBM QR#14
RDF-3X	1.65	14.88	1640
BigOWLIM	138	5.7	3320
Jena TDB	3.52	2.18	2998
WaterFowl Mode 2	1.80	10.18	1710
WaterFowl Mode 3	1.75	10.13	1680

high selectivity, complex 'triangle' query pattern and large input with low selectivity. Query #2 is performed more rapidly by Jena TDB and BigOWLIM but WaterFowl is faster than RDF-3X. We consider that this is due to a better consideration of this query particular pattern. It highlights that our query optimization has room for improvement.

Table 3. Inference-based query answering times (sec) on univ100

	QR#4	QR#5	QR#6	QR#7	QR#10
RDF-3X	4.2	2.5	15.3	1.4	1.6
OWLIM-SE	705	16771	72	1708	3.65
Jena TDB	4.85	6.3	30.7	207	1.55
WaterFowl Mode 2	3.66	2.3	13.4	1.2	1.4

In the context of queries requiring RDFS entailment, we are testing RDF-3X with query rewriting performed using a DL reasoner against our system. That is, we have implemented a simple RDFS query rewriting on top of RDF-3X which generates SPARQL queries with `UNION` clauses. The RDF-3X approach enables to perform query rewriting in the context of the considered fastest RDF Store. Note that the two other systems do not require this machinery since they rely on a materialization approach. Table 3 highlights that our system slightly outperforms the inference-enable RDF-3X on a set of five distinct LUBM queries, requiring different forms of reasoning, *i.e.*, based on concept and property subsumption relationships. It has already been emphasized that due to its large number of indexes, RDF-3X is very competitive or even faster than some materialization-based systems. Due to our ontology elements encoding with prefix enabled navigation and minimalist materialization of *rdfs:domain* and *rdfs:range* information, we outperform all systems on these five queries.

6 Conclusion

We have designed and implemented a novel type of immutable 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.*, RDF-3X. We consider that this is quite a strong encouragement toward pursuing our work on WaterFowl. We consider that this is due to the advantage of our highly compressed data and implementing all data retrieving operations on SDS functions, *i.e.*, *access*, *rank*, *select* and their prefix counterparts. We also believe that 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 investigations will include the distribution of triples over a cluster of machines and the support for updates in both the TBox and the ABox, which is not trivial since actual wavelet trees do not support efficient updates.

References

1. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable semantic web data management using vertical partitioning. In: VLDB, pp. 411–422 (2007)
2. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. In: WWW, pp. 41–50 (2010)
3. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
4. Fernández, J.D., Martínez-Prieto, M.A., Gutierrez, C.: Compact representation of large RDF data sets for publishing and exchange. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 193–208. Springer, Heidelberg (2010)
5. Goasdoué, F., Manolescu, I., Roatis, A.: Efficient query answering against dynamic RDF databases. In: EDBT, pp. 299–310 (2013)
6. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: SODA, pp. 841–850 (2003)
7. Grossi, R., Ottaviano, G.: The wavelet trie: maintaining an indexed sequence of strings in compressed space. In: PODS, pp. 203–214 (2012)
8. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for OWL knowledge base systems. J. Web Sem. 3(2-3), 158–182 (2005)
9. Martínez-Prieto, M.A., Gallego, M.A., Fernández, J.D.: Exchange and consumption of huge RDF data. In: ESWC, pp. 437–452 (2012)
10. Munro, J.I.: Tables. In: FSTTCS, pp. 37–42 (1996)
11. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. 19(1), 91–113 (2010)
12. Pérez-Urbina, H., Horrocks, I., Motik, B.: Efficient query answering for OWL2. In: ISWC, pp. 489–504 (2009)
13. Rodriguez-Muro, M., Calvanese, D.: High performance query answering over DL-lite ontologies. In: KR (2012)
14. Rosati, R., Almatelli, A.: Improving query answering over DL-lite ontologies. In: KR (2010)
15. Tsialiamanis, P., Sidirourgos, L., Fundulaki, I., Christophides, V., Boncz, P.A.: Heuristics-based query optimisation for SPARQL. In: EDBT, pp. 324–335 (2012)