REDD: An Algorithm for Redundancy Detection in RDF Models

Floriana Esposito, Luigi Iannone, Ignazio Palmisano, Domenico Redavid, and Giovanni Semeraro

Dipartimento di Informatica,
Università degli Studi di Bari,
Campus, Via Orabona 4, 70125 Bari, Italy
{esposito, iannone, palmisano, d.redavid, semeraro}@di.uniba.it

Abstract. The base of Semantic Web specifications is Resource Description Framework (RDF) as a standard for expressing metadata. RDF has a simple object model, allowing for easy design of knowledge bases. This implies that the size of knowledge bases can dramatically increase; therefore, it is necessary to take into account both scalability and space consumption when storing such bases. Some theoretical results related to blank node semantics can be exploited in order to design techniques that optimize, among others, space requirements in storing RDF descriptions. We present an algorithm, called REDD, that exploits these theoretical results and optimizes the space used by a RDF description.

1 Motivation

The realization of the Semantic Web (SW) vision [1] needs ontologies for generating or interpreting (semantic) metadata for resources. It is fundamental to have ontology creation and integration steps in order to share structural knowledge between ontology designers and users. Ontologies are to be expressed in RDF according to SW specifications, using languages such as RDFS[1] and OWL.[2] It is important to note that both RDFS and OWL ontologies can be expressed as RDF graphs, so that ontologies can be treated exactly as other RDF models. In RDF design, the least power principle was applied: data structures are to be kept as simple as possible. This imposes to have very simple basic components, that are URIs[3], blank nodes and statements (or triples). These design decisions have the drawback that RDF descriptions tend to grow fast as the complexity of the knowledge they represent increases. This observation encourages SW research to investigate toward the most effective storage solutions for RDF knowledge bases, in order to minimize required space. Intuitively, the lesser the number of triples a software (say, a query engine) has to examine, the faster it will process them.

[1] http://www.w3c.org/TR/rdf-schema
[2] http://www.w3c.org/2004/OWL
[3] http://www.w3.org/Addressing/

This issue has already been deeply investigated, as reported in the section 2.2; recently, some theoretical results were issued by both W3C in [6] and by Gutierrez et al. in [4]. Actually, these results apply also to RDFS, but in this paper we will refer only to blank node semantics. Relying on these results, we developed an algorithm to detect redundancies introduced by blank nodes in a RDF Description. Such redundancies can be removed by mapping blank nodes into concrete URIs or into different blank nodes, without changing or diminishing the RDF graph semantics. In other words, some descriptions can be expressed with lesser triples with no semantic loss.

Moreover, redundancy detection can turn out to be useful in higher level tasks, such as ontology design and alignment. Let us suppose to have designed some classes (say in OWL) and let one of them be a cardinality restriction. If somewhere in the ontology it has a name (an URI), as depicted in Figure 1 (i.e.: ns:Test), and somewhere else we created the same restriction without using a name (so using an anonymous restriction class), we would have defined this class twice unnecessarily, so intuitively we introduced redundancy. This kind of repetitions can be detected thanks to blank node semantics and removed, thus simplifying the design of the ontology. The same situation occurs, obviously, if both the restrictions are represented by blank nodes.

Another situation in which the algorithm can be useful is in ontology importing, i.e. the use of the *owl:imports* directive. In this case, let *A* and *B* be two ontologies, and *A owl:imports B*; referring to the restriction example, if there is an anonymous restriction in *B*, and *A* needs the same restriction, the designer of *A* needs to define its own restriction (since a blank node cannot be identified from outside the model in which it is defined). From the OWL point of view, however, *A* contains every statement in *B*, so the complete model (i.e. the model containing *A* plus the import closure) is redundant. The problem can become serious if there are multiple *owl:imports*. Suppose an ontology is imported more than once, e.g. *A owl:imports B, C* and *B, C owl:imports D*; in this latter case, *D* is

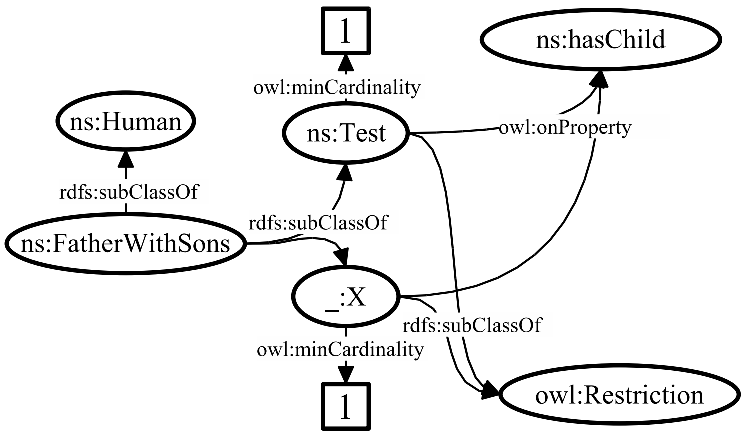

Fig. 1. Example of redundant Restrictions

imported twice, and this means that, unless the code used to resolve *owl:imports* handles the case of multiple imports, every blank node in D is duplicated in the resulting ontology. We will see an example of this situation in Section 5.

In order to accomplish this, we will show a *correct* algorithm (in the sense that it produces descriptions equivalent to the starting ones) without claiming for its *completeness* (it is not guaranteed to find the minimal equivalent description) called REDD (REDundancy Detection). This algorithm has been integrated in our RDF management system, named RDFCore [2].

The remainder of this paper is organized as follows: Section 2 presents some necessary notions on RDF semantics, together with a brief survey of related work on RDF storage. In Section 3, the REDD algorithm is illustrated in detail; Section 4 describes RDFCore, the system in which we implemented the REDD algorithm. Some experimental results are presented in Section 5.

2 Preliminaries

2.1 Basic Notions

We collect here some definitions and theorems that will be useful in the rest of the paper. Most of them have been taken from [6] and [4] and recalled here to make this paper as self-contained as possible. However, we assume the reader familiar with RDF Concepts and Syntax:[4]

Definition 1 (RDF-Graph). *A RDF-Graph is a set of RDF statements. Its nodes are URIs, literals or blank nodes (identifiable nodes with no intrinsic names*[5]*) representing subjects and objects of the statements. Its edges are labeled by URIs and represent the predicates of the triples.*

A small example can be found in Figure 2.

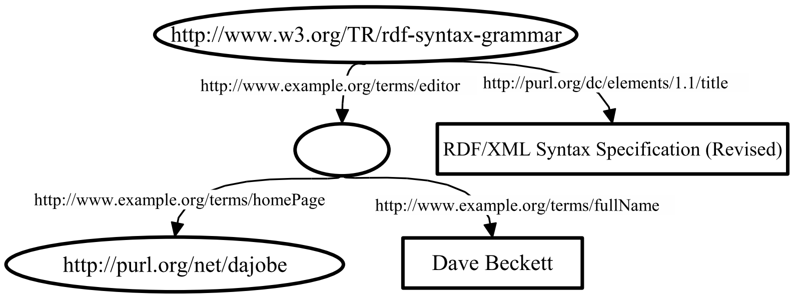
http://www.w3.org/TR/rdf-syntax-grammar
http://www.example.org/terms/editor
http://purl.org/dc/elements/1.1/title
RDF/XML Syntax Specification (Revised)
http://www.example.org/terms/homePage
http://www.example.org/terms/fullName
http://purl.org/net/dajobe
Dave Beckett

Fig. 2. A small example from http://www.w3.org/TR/rdf-syntax-grammar/

[4] `http://www.w3.org/TR/rdf-concepts/`
[5] `http://www.w3.org/TR/rdf-concepts/#section-URI-Vocabulary`

Definition 2 (Mapping). *Let N be a set of URIs, blank node names and literals. A mapping is a function $\mu : N \rightarrow N$ that changes a node name into another one.*

Definition 3 (Instance). *Let μ be a mapping from a set of blank nodes to some set of literals, blank nodes and URIs and G a graph, then any graph obtained from G by replacing some or all of the blank nodes N in G by $\mu(N)$ is an instance of G.*

Definition 4 (Lean Graph). *A RDF graph is lean if it has no instance which is a proper subgraph of the graph.*

The following results are proved in [6]:

Lemma 1 (Subgraph Lemma). *A graph entails all its subgraphs.*

Lemma 2 (Instance Lemma). *A graph is entailed by any of its instances.*

This means that every non-lean graph is equivalent to its *unique* lean subgraph [4]. Relying on these notions, in Section 3 we will present an algorithm that reduces non-lean graphs under certain conditions.

2.2 Related Work

Effective storage of RDF has always been bound to another key issue: Querying models. This was because no recommendation, at the time of writing, has been completed by W3C for RDF description querying (SPARQL[6] is at the Working Draft stage of its evolution); thus, different solutions were developed, each one with its own query language and related optimizations. Some members of RDF Data Access Group issued a report[7] in which six query engines were examined aiming to compare different expressive power of the underlying query languages. Actually, many different triple storage strategies are available. Among the systems implementing them, we remark the toolkit from HP Semantic Web Lab, called Jena [8, 9]. At the time of writing, Jena supports RDQL as query language, with support for SPARQL in a separate project, ARQ.[8]

Other interesting approaches to RDF data model optimization relies on properties of the RDF graph. One of them is described in [5], where the authors present an approach based on an intermediate layer between application data structures and the abstract triple syntax that uses hypergraphs. In this approach, theoretical results on graph theory can be used to minimize graphs and to cast application requests to well known graph problems, thus allowing to optimize different usage scenarios for RDF graphs.

[6] `http://www.w3.org/2001/sw/DataAccess/rq23/`

[7] `http://www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query/rdfquery.pdf`

[8] `http://cvs.sourceforge.net/viewcvs.py/jena/ARQ/`

3 Redundancy Detection

3.1 REDD Algorithm

Our redundancy detection algorithm is based on the notion of lean subgraph of a RDF graph. The lean subgraph is a subset of the RDF graph, and, as a consequence, is a subset of the set of statements of the original graph, having the property of being the smallest subgraph that is instance of the original graph. A pseudo code version of it can be found in Figure 3. The output of this algorithm has as a requirement the characteristic of expressing the same content of the original RDF graph (though in a more compact way). The output can be obtained from the original graph leaving untouched the ground part of the graph

```
ConnGraph{Set blanks, Model submodel, Map bToVarNames}

Set FindRedundancies(Model m){
  Set redundancies
  Set connGraphs =
      CreateConnGraphs(m)
  FOR EACH graph in connGraphs{
    Query q = CreateQuery(graph)
    Set redundancy = ExecuteQuery(m,q)
    ADD redundancy to redundancies
  }
  RETURN redundancies
}

Query CreateQuery(ConnGraph g){
  Query q
  FOR EACH s in g.statements{
    IF (s.subj is blank) AND
        (s.subj is not in g.bToVarNames){
      create a variable name vn
      PUT(g.bToVarNames, s.subj, vn)
      ADD vn to q
    }ELSE{ vn = s.subj }
    IF (s.obj is blank) AND
        (s.obj is not in g.bToVarNames){
      create a variable name o
      PUT(g.bToVarNames, s.obj, o)
      ADD o to q
    }ELSE{ o = s.obj }
    ADD (s, s.pred, o) condition to q
  }
  RETURN q
}
```

```
Set CreateConnGraphs(Model m){
  Set cg
  Map blanksTocg
  FOR EACH s in m{
    IF s.subj is blank{
      IF exists g in blanksTocg
          mapped by s.subj{
        add s to g
      }ELSE{
        create g for s.subj
        add s to g
        put g in cg
        PUT(blanksTocg, s.subj, g)
      }
      IF s.obj is blank{
        add o to g.blanks
        PUT(blanksTocg, o, g)
      }
    }
  }
  RETURN cg
}

Set ExecuteQuery(Model m, Query q){
  Bindings b = QUERYON(m, q)
  Set redundancy
  FOR EACH binding in b{
    PUT binding.values in redundancy
  }
  RETURN redundancy
}
```

Fig. 3. Pseudo-code description of the REDD algorithm

(i.e. every node that is not blank and any edge connecting non-blank nodes), and mapping from blank nodes to nodes already existing in the graph (blank nodes or URIs). The result is bound to be a subset of the original graph, apart from the identifiers of blank nodes.

Our algorithm searches for redundant blank nodes by looking at the graph and trying to find blank nodes that do not contain any additional information w.r.t. other nodes in the graph. Therefore, a blank node *b* is redundant if there is a node *n* that is involved in a set of statements that would be equal to the set of statements involving *b* if we replaced the occurrences of *b* with occurrences of *n*.

This is a special case of a more general view: taking as reference a subgraph built up of statements with blank nodes as subject and object, it is possible to search for a different subgraph of the model, isomorphic to the given subgraph (i.e. with the same properties between the nodes). On these two graphs, the algorithm can be applied considering the set of edges minus the edges already considered in the graph.

Our approach consists in finding a mapping from the original blank nodes of the graph to URI in the graph or to different blank nodes already in the graph (i.e. we do not introduce any new blank node). As an example, let us consider a simple graph containing two statements, say:

```
  _:X ns:aGenericProperty ns:b
ns:a ns:aGenericProperty ns:b
```

we can determine that the graph is not lean by considering the mapping

$$\_ : X \rightarrow ns : a$$

The result is a graph with a single statement

```
ns:a ns:aGenericProperty ns:b
```

which is lean by definition (being a graph with no blank nodes).

More formally, called:

- *ORIGIN* the original graph
- *RESULT* the new graph we are going to build
- *X* the anonymous node we want to map

we define:

Definition 5 (SUBMODEL). *All the statements in* ORIGIN *in which* X *is the subject.*

Definition 6 (SUPERMODEL). *All the statements in* ORIGIN *in which* X *is the object.*

We then can check every possible mapping from *X* to an URI or to a blank node identifier already occurring in *ORIGIN* for applicability to obtain an instance of *ORIGIN* which is both an instance and a proper subgraph (an approximation

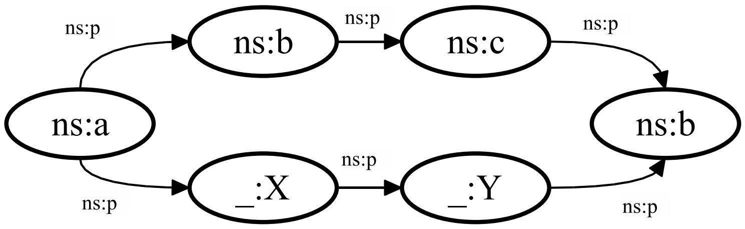

Fig. 4. Chained redundant blank nodes

of the lean subgraph) simply by checking that *SUBMODEL* of X is contained in *SUBMODEL* of the candidate node and *SUPERMODEL* of X is contained in *SUPERMODEL* of the candidate node. In fact, it can be easily proved that such a mapping does not produce any statement not contained in *ORIGIN*; *RESULT* then is a graph containing the same ground statements and a subset of the statements containing blank nodes. The missing statements are those containing the X node we just mapped. From the logical point of view, the information expressed by the graph is unchanged, since the mapping is equivalent to changing from:

$$\exists X.p(X,b) \text{ and } \exists a.p(a,b)$$

to

$$\exists a.p(a,b)$$

which are equivalent, not being stated that X is different from a. This mapping can be built for every redundant blank node in *ORIGIN*, but in some situations it is not guaranteed to find all redundancies. In fact, as in Figure 4, it is possible to have a chain of redundant blank nodes which cannot be spotted with a one-level visit of the RDF graph. In fact, in Figure 4, the two blank nodes represent the same structure as the two nodes labeled b and c. To find this redundancy, it is necessary to switch from a single node view to a multi node view.

For ease of reference, let us use an N-TRIPLE-like notation for the example in Figure 4:

```
ns:a ns:p _:X
ns:a ns:p ns:b
ns:b ns:p ns:c
ns:c ns:p ns:d
_:X ns:p _:Y
_:Y ns:p ns:d
```

considering the subgraph

```
_:X ns:p _:Y
```

named *BLANKS* for future reference, its structure can be described with a query like (using RDQL as query language, for the example)

```
SELECT ?a, ?b WHERE (?a, ns:p, ?b)
```

This query offers two results when executed against the model in the example: the first result is the mapping $?a \rightarrow \_ : X$ and $?b \rightarrow \_ : Y$, while the second is $?a \rightarrow ns : b$ and $?b \rightarrow ns : c$. Actually, the results are two graphs; checking every incoming edge and outgoing edge of the two graphs, we can determine if the graphs are equivalent, in analogy with the previous particular case in which the graph degenerates to a single blank node. This can be done adding conditions to the original query:

```
SELECT ?a, ?b WHERE (?a, ns:p, ?b)(ns:a ns:p ?a)(?b ns:p ns:d)
```

This can be done in a general way starting from the subgraph built considering every triple involving the *BLANKS* subgraph, and then generating the query with a condition for every triple and a variable for each blank node, keeping track of the blank node → variable name association. The resulting query, executed on the model, will surely produce one resulting graph (the subgraph used to generate it); any other result is a graph that respects the constraints we imposed on the single node case: on any node, the set of incoming edges includes the set of incoming edges of the corresponding node in the *BLANKS* graph (the corresponding node is the node that grounds the same variable), excluding the edges already included in the graph (i.e. where both subject and object are variables in the query).

For ease of future reference, we give the following definition:

Definition 7 (CONNECTED SUBGRAPH). *A connected subgraph of a graph G is a subgraph of G containing at least one blank node (as subject or object); if more than a blank node is contained in the graph, then the blank nodes make up a chain (i.e. it is possible to navigate from a blank node to another following the predicates). The connected subgraph is made up of all the triples in G that involve at least one of these blank nodes.*

As an example, in Figure 4 there is one chain of blank nodes; the corresponding connected graph is:

```
ns:a ns:p _:X
 _:X ns:p _:Y
 _:Y ns:p ns:d
```

This algorithm has been implemented in two steps: first, the special case in which we consider only single blank nodes (i.e. no chain redundancies are detected) has been implemented both as a Java class (built on top of the Jena API) and directly in the storage layer of RDFCore with stored procedures in the Oracle DB. The second step, i.e. the implementation of both single nodes and chains detection, has been completed as a Java class (again using the Jena API), and at the time of writing we are implementing it in the storage layer.

3.2 REDD Computational Complexity

In this subsection we will shortly carry out an *a priori* evaluation of computational cost required by REDD algorithm. We will keep as reference the pseudo code version of REDD in Figure 3. Obviously, the actual implementations working both in memory and natively on the storage layers (see Section 4) underwent some optimizations, not shown in the pseudo code for sake of brevity; hence calculations in this section represent only an upper theoretical limit for the computational cost of REDD. In section 5, the reader can find some empirical evaluations.

We start defining some metrics on RDF descriptions on which, as shown below, REDD complexity depends.

Definition 8 (RDF Description metrics). *Be G a RDF description and n a generic node in G then*

- N_T^G *stands for the number of RDF triples within G*
- $N_{T_B}^G$ *stands for the number of RDF triples within G containing at least a blank node*
- $N_{T_{NB}}^G$ *stands for the number of RDF triples within G with no blank nodes (it is equal to $N_T^G - N_{T_B}^G$)*
- $\#_{CG}^G$ *stands for the number of connected subgraphs with blank nodes within G*

Referring to Figure 3, complexity of *FindRedundancies* C_{FR} is:

$$C_{FR} = C_{CCG} + \#_{CG}^G * C_{CQ} + \#_{CG}^G * C_{EQ} \tag{1}$$

where

- C_{CCG} is the complexity of the *CreateConnGraphs* operation, which is $O(N_T^G)$ (linear in the size of the model); more in detail, the main cycle of *CreateConnGraphs* depends on N_T^G, while each operation executed in the cycle does not depend on N_T^G (depending on the implementation, map and set operations can vary their complexity; assuming an hash implementation for both of them, every operation can be considered $O(1)$). On the other hand, the number of mappings in the *blanksTocg* map depends on the degree of connectedness in the graph: at worse, there will be no more than N_T^G mappings (because the number of mappings cannot exceed the number of triples in the graph), in the case that every statement has at least a blank node as subject or object;
- C_{CQ} is the complexity of the *CreateQuery* operation, which depends on the number of triples in the connected subgraph it is operated onto; since the connected subgraphs are disjoint set (if two connected subgraphs overlap, i.e. they have some common blank node and in consequence some common triple, they are actually one connected subgraph by definition, and they are built in this way), the whole group $\#_{CG}^G * C_{CQ}$ has complexity $O(N_{T_B}^G)$;
- C_{EQ} is the complexity of the *ExecuteQuery* operation, which depends on the *QUERYON* operator and on the number of results the query finds in the model. In the worst case (very unlikely to happen), the number of results can be equal to the number of resources in the model, which is at worst

$3 * N_T^G$ (again, very unlikely to happen - in particular, since in the query the predicate URI is never a variable, if the second situation is the case then the number of results will be exactly one, and no redundancies will be possible). Hence, this portion is $O(N_T^G)$. The *QUERYON* operator complexity depends on the query facility, that in the actual implementation relies on Jena RDQL support. An upper limit for the complexity of this operation can be calculated considering each condition in the query and verifying them one by one against the model. There is a condition for each statement in the connected graph, and each one of them requires (in an implementation with no optimizations) at most N_T^G checks on the model; under these assumptions (which are surely an underestimation of the real performances), the whole group $\#_{CG}^G * C_{EQ}$ has complexity $O(\#_{CG}^G * N_T^G + N_T^{G2})$

As a result, C_{FR} has complexity $O(N_T^{G2})$; in particular, the quadratic complexity depends on the query phase of the algorithm, since the other two main sections are $O(N_T^G)$, and it comes from a deliberate overestimation of the query performances. As an example, in an hypothetical implementation a simple indexing on statement predicates P reduces the complexity of the search from $O(N_T^G)$ to $O(\#_P)$. In real models, it can reasonably be assumed that $\#_P << N_T^G$.

In the next sections we will briefly present the RDFCore system, where REDD has been implemented, and, in section 5, we will show some empirical results for the algorithm.

4 The RDFCore Component

The RDFCore component, presented in [2, 7], is based on two classes, *DescriptionManager* and *TripleManager*, and an interface, *RDFEngineInterface.*

RDFEngineInterface is an interface that enables the managers to abstract from the physical persistence details. Thanks to this design, based on the well known *Strategy* pattern [3], the system can use one or more persistence implementations (with different performance or scalability tradeoffs) whenever needed, on the basis of the needs of the external applications, without the programmer having to bother about different APIs. Currently, there are four implementations of *RDFEngineInterface*, two based on the well-known Jena Semantic Web Toolkit, one with MySQL RDBMS [9] and another with SQL Server[10] as persistent storage; a third implementation is based on RDF/XML files. The last implementation, called *RDFEngineREDD*, is the one in which we embedded the REDD algorithm natively in the storage level. It uses Oracle[11] as RDBMS. The database has been chosen because of the availability of stored procedures and

[9] `http://dev.mysql.com/doc/mysql/en/index.html`

[10] `http://www.microsoft.com/sql/`

[11] Oracle 9.2.0.1.0 (Oracle 9i Release 2) `http://otn.oracle.com/documentation/oracle9i.html`

the ability to execute Java code directly on the database, avoiding the overhead of data transfer that would have arisen using different solutions.

Currently RDFCore is a central component within the core infrastructure of the software architecture that will result out of the 6^{th} Framework Project VIKEF (Virtual Information and Knowledge Framework Priority 2.3.1.7. Semantic Based Knowledge Systems Contract no.: 507173).

5 Experimental Results

To evaluate the scalability of our implementation of the REDD algorithm in the *RDFEngineREDD* implementation of *RDFEngineInterface*, we built a set of Models to check some situations inspired by real models; the results are in Table 3. The models come from different sources: the first two, *lean* and *nolean*, are from [6], where they are presented as basic examples of lean and non-lean graphs. *nolean2B* is a slight variation of *nolean*, with two redundant blank nodes. *cycleTest* is used to check the behavior of the algorithm when dealing with complex cyclic situations in graphs, while *blankChain* contains a chain of redundant blank nodes like in the Figure 4. *restriction* contains a redundant restriction class definition (as in Figure 1) together with a redundant union class definition (in OWL); the last Model, *daml*, contains a sketch of a DAML ontology, with some class definitions including both Restriction, Union and Intersection types. For each model, we recorded the number of statements, the number of blank nodes present in the graph, the elapsed time to insert the models in our persistence (in milliseconds), the elapsed time to execute REDD (in milliseconds) and the number of removable blanks in the graph. Since the size of these models is way too small to evaluate scalability on model size and complexity, we kept these test cases as correctness checks while developing the algorithm, and then created a parametric method to generate bigger models with known structure, in order to scale the size and complexity without having to check the correctness of the results (which can be a time consuming task for models with more than some tens of nodes). The parameters we used are: the number of blank nodes in a graph, the number of incoming/outgoing edges for each node, and the number of redundancies for each blank node (i.e. a blank node can be found redundant with one or more nodes in the graph). The test models were built scaling on the three parameters independently (showed in Table 1); in the last section, both the number of blank nodes and the number of redundancies per node is augmented.

These tests were performed on the database implementation, that, as said earlier in the paper, still does not handle the blank node chains.

In order to give a preliminary evaluation of the complete algorithm, we used some models built from a real ontology, as said in Section 1. The ontology is the BM Ontology[12], that aims to describe the domain of business process description. In this ontology, we tried to artificially increase the number of blank nodes and of blank nodes chain.

[12] `http://www.bpiresearch.com`

Table 1. Fake models scaling on ingoing / outgoing edges, blank node number and redundancy number

Model id	Triple #	Blank node #	Storing time (ms)	REDD (ms)	Redundancies #	Removable blanks #	Ingoing/ outgoing edges
0	120	1	1469	62	5	1	10
1	240	1	2469	94	5	1	20
2	360	1	3438	141	5	1	30
3	480	1	4515	188	5	1	40
4	600	1	5266	234	5	1	50
5	720	1	6328	297	5	1	60
6	840	1	7109	360	5	1	70
7	960	1	8172	437	5	1	80
8	1080	1	9203	594	5	1	90
9	1200	1	11016	625	5	1	100
10	200	5	1953	78	1	5	10
11	400	10	3766	125	1	10	10
12	600	15	5406	250	1	15	10
13	800	20	7203	219	1	20	10
14	1000	25	10000	281	1	25	10
15	1200	30	10860	375	1	30	10
16	1400	35	12828	407	1	35	10
17	1600	40	14844	469	1	40	10
18	1800	45	15969	563	1	45	10
19	2000	50	18047	750	1	50	10
20	120	1	2235	453	5	5	10
21	220	1	2235	93	10	10	10
22	320	1	3188	156	15	15	10
23	420	1	3828	188	20	20	10
24	520	1	4485	234	25	25	10
25	620	1	5047	266	30	30	10
26	720	1	5813	297	35	35	10
27	820	1	6907	546	40	40	10
28	920	1	7360	406	45	45	10
29	1020	1	8188	437	50	50	10
30	600	5	4906	234	5	5	10
31	2200	10	18328	922	10	10	10
32	4800	15	39141	2187	15	15	10
33	8400	20	69578	4203	20	20	10
34	13000	25	118031	6078	25	25	10
35	18600	30	171563	10031	30	30	10

Our use of the BM Ontology was as follows: we loaded the ontology in a Jena OntModel, with no reasoning in order to use only the original triples, and then wrote out the complete model, obtaining a RDF model containing the BM Ontology and the import closure. Then, we reloaded this new model (that we will call BMO1) in an OntModel. The resolution of *owl:imports* directive in this admittedly pathological model produces a new inferred model in which every blank node and blank node chain is duplicated, and this produces redundancies that REDD can discover (model BMO2). We repeated the procedure and obtained the models BMO3 and BMO4. The results are shown in Table 2.

As can be seen in Table 1, the insertion of new descriptions in RDFCore roughly scales linearly with the size of the descriptions. The performance overhead due to index updating, however, increases when the number of triples in a description increase, so the total complexity is more than linear. The heavy indexing, on the other side, enables us to obtain very good results when running

Table 2. Models with blank node chains

Model id	Triple #	Blank node #	Redundant triples #	REDD (ms)	Chain #
BMO1	12267	1416	16	3294	804
BMO2	16487	2832	8440	5258	1608
BMO3	20707	4248	12660	18146	2412
BMO4	24927	5664	16880	102648	3216

Table 3. Some real-world models tests

Model id	Triple #	Blank node #	Storing time (ms)	REDD (ms)	Removable blanks #
lean	2	1	140	32	0
nolean	2	1	62	31	1
nolean2B	3	2	46	47	2
blankChain	7	2	94	31	0
cycleTest	15	2	204	31	1
restriction	35	17	500	93	7
daml	38	33	718	282	16

the REDD algorithm on the data. About the real size reduction of the model after the removal of the blank nodes (which means the removal of every triple referring to these nodes), it is not possible to draw general conclusions since the number of triples strongly depends on the graph; the only reasonable lower limit is two triples per blank node, since it is quite unusual to have a dangling blank node or a graph rooted in a blank node, and in these cases it is unlikely that the nodes are redundant (e.g. `ns:a ns:aProperty _:X` means that *ns:a* has a filler for the role *ns:aProperty*, but nothing else is known about this filler; adding another statement, `ns:a ns:aProperty _:Y`, would assert the same thing; unless stating that *_:X* is different from *_:Y*, REDD signals the nodes as redundant).

About the implementation running in memory, the test data shows an interesting behavior: while BMO1 only contains 16 redundant triples, BMO2 contains 8440 redundant triples. What happens here is that the BMO1 model contains 4 redundant restrictions (that are anonymous cardinality restrictions similar to the one represented in Figure 1), each built up of 4 triples; the other 1412 blank nodes are arranged in 800 chains (with different chain length), usually RDF lists, that are not redundant with any structure in BMO1. In BMO2, however, the duplication of these structures produces 1608 blank node chains, and each one of them is redundant with at least one structure. Same explanation for the further increase in BMO3 and BMO4.

From this information, it is possible to infer what would be the results of querying a reduced model: in fact, BMO1 is only 12 triples bigger than the smallest model that REDD can produce. Since every blank node chain produces a RDQL query to be executed on the model, from the data it is possible to deduce that the time required for a query on BMO4 is bigger than the time required for a query in BMO1 (from about 4 ms in BMO1 to more than 26 ms in BMO4). This huge difference between query performance can be partially attributed to lack of optimization in the current algorithm implementation (e.g., two redundant blank node chains produce two queries, but the queries are equal; this is recognized only in some cases by our implementation), but it is an empirical confirmation of our initial intuitive claim that queries on a smaller model are faster than queries on a larger model.

Our aim in the ongoing work (i.e. pushing down into the persistence layer the chain redundancy detection) is to match the performances of the first version of the algorithm. Moreover, we plan extensions to the algorithm applications, e.g. recognition of Alt and Bag structures in order to be able to detect duplications. Another extension (based on OWL semantics) is the recognition of the use of Lists in the declaration of union and intersection classes; while differently ordered lists are different at the RDF level, they express the same meaning at the OWL level, and this should be detected as redundancy. Also, it is necessary to establish ordering criteria when choosing the blank nodes to be removed from the graph: in fact, detecting a redundancy corresponds to finding of set inclusion relationships between *SUPERGRAPHS* and *SUBGRAPHS*; the choice can be made freely only when *SUPERGRAPHS* and *SUBGRAPHS* are equal, while in other cases an ordering criterion must be used.

6 Conclusions

In this paper we started from the consideration that SW is based on a particular language for metadata description, RDF, whose semantics has been recently thoroughly investigated by W3C and other researchers. This initial effort produced some valuable results in terms of theoretical foundations for entailment in RDF. We examined, in particular, results concerning blank node semantics and their effects on the problem of compacting RDF graphs. We presented a correct algorithm for spotting out redundant blank nodes in RDF graphs and we provided a pseudo code implementation. We discussed its complexity proving it is tractable (polynomial). Afterward we presented its actual prototypical implementation within our RDF management system (RDFCore). From empirical evaluation we found out that these initial results are encouraging (being it a prototype). Furthermore, redundancies can be also referred to a vocabulary. In fact this work did not take into account RDFS (and its derivatives) semantics that can be deeply exploited for compacting descriptions.

Acknowledgments

This research was partially funded by the European Commission under the 6th Framework Programme IST Integrated Project VIKEF - Virtual Information and Knowledge Environment Framework (Contract no. 507173, Priority 2.3.1.7 Semantic-based Knowledge Systems; more information at `http://www.vikef.net`), and under the DELOS 2 Network of Excellence on Digital Libraries started on January 1, 2004 Priority IST-2002-2.3.1.12 Technology-enhanced Learning and Access to Cultural Heritage - Contract no.: G038-507618 (`http://www.delos.info/`).

References

1. Berners-Lee, T.: Semantic Web Road map (1998) `http://www.w3.org/DesignIssues/Semantic.html`.
2. Esposito, F., Iannone, L., Palmisano, I., Semeraro, G.: RDF Core: a Component for Effective Management of RDF Models. In Cruz, I.F., Kashyap, V., Decker, S., Eckstein, R., eds.: Proceedings of SWDB'03, The First International Workshop on Semantic Web and Databases, Co-located with VLDB 2003, Humboldt-Universität, Berlin, Germany, September 7-8, 2003. (2003)
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. 1st edn. Addison-Wesley (1995)
4. Gutiérrez, C., Hurtado, C., Mendelzon, A.O.: Foundations of Semantic Web Databases. In: Proceedings of ACM Symposium on Principles of Database Systems (PODS) Paris, France, June 2004. (2004)
5. Hayes, J., Gutiérrez, C.: Bipartite graphs as intermediate model for rdf. In: International Semantic Web Conference. (2004) 47–61
6. Hayes, P.: RDF semantics (2004) W3C Recommendation 10 February 2004 `http://www.w3.org/TR/rdf-mt/`.
7. Iannone, L., Palmisano, I., Redavid, D.: Optimizing RDF storage removing redundancies: an algorithm. In Ali, M., Esposito, F., eds.: Proceedings of the 18th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Bari, Italy, June 22-25 2005. Lecture Notes in Artificial Intelligence, Springer (2005) (to appear).
8. McBride, B.: JENA: A Semantic Web toolkit. IEEE Internet Computing **6** (2002) 55–59
9. Wilkinson, K., Sayers, C., Kuno, H.A., Reynolds, D.: Efficient RDF storage and retrieval in jena2. In Cruz, I.F., Kashyap, V., Decker, S., Eckstein, R., eds.: Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases, Co-located with VLDB 2003, Humboldt-Universität, Berlin, Germany, September 7-8, 2003. (2003) 131–150