Managing RDF Metadata for Community Webs*

Sofia Alexaki Vassilis Christophides Greg Karvounarakis Dimitris Plexousakis
ICS-FORTH, Vassilika Vouton, P.O.Box 1385, GR 711 10, Heraklion, Greece
{alexaki, christop, gregkar, dp}@ics.forth.gr
Karsten Tolle
University of Hannover, Welfengarten 1, 30167 Hannover, Germany
tolle@uni-hannover.de
Bernd Amann Irini Fundulaki Michel Scholl Anne-Marie Vercoustre
INRIA Rocquencourt, 78153 Le Chesnay Cedex, France
{amann, fundulak, scholl, vercoust}@cosmos.inria.fr

Abstract

The need for descriptive information, i.e., metadata, about Web resources has been recognized in several application contexts (e.g., digital libraries, portals). The Resource Description Framework (RDF) aims at facilitating the creation and exchange of metadata, as directed labeled graphs serialized in XML. In particular, the definition of schema vocabularies enables the interpretation of semistructured RDF descriptions using taxonomies of node and edge labels. In this paper, we propose (i) a formal model capturing RDF schema constructs; (ii) a declarative query language featuring generalized path expressions for taxonomies of labels (iii) a metadata management architecture for efficient storage and querying of RDF descriptions and schemas.

1 Introduction

Metadata are widely used in order to fully exploit information resources (e.g., sites, documents, data, etc.) available on the WWW [?]. Indeed, metadata permit the description of the content and/or structure of WWW resources in various application contexts: digital libraries, infomediaries, enterprise portals, etc. The Resource Description Framework (RDF) [?] aims at facilitating the creation and exchange of metadata, as any other Web data. More precisely, RDF descriptive (meta)data are represented as *directed labeled graphs* (where nodes are called *resources* and edges are called *properties*) which are serialized using an XML syntax. Furthermore, RDF schema [?] vocabularies are used to define the labels of nodes (called *classes*) and edges (called *property types*) that can be used to describe and query resources in specific user communities.[1] These labels can be organized into appropriate taxonomies, carrying inclusion semantics. In this paper, we are focusing on the design of a metadata management system for storing and querying both RDF descriptions and schemas as semistructured data [?].

Our work is motivated by the fact that existing semistructured models (e.g., OEM [?, ?], YAT [?, ?]) cannot capture the semantics of node and edge labels provided by RDF schemas

*This work was partially supported by the European project C-Web (IST-1999-13479).

[1] A community Web is determined by an RDF schema and a number of information resources that its members want to share.

(i.e. taxonomies of classes and property types), while semistructured or XML query languages (e.g., LOREL [?], UnQL [?], StruQL [?], XML-QL [?], XML-GL [?]) are not suited to exploit taxonomies of labels for query evaluation and optimization (i.e., pattern vs. semantic matching of labels). In this context, we propose a declarative query language for RDF, called RQL. RQL relies on a graph data model allowing us to (partially) interpret semistructured RDF descriptions by means of one or more RDF schemas. Thus, RQL adapts the functionality of semistructured query languages to the peculiarities of RDF but also extends this functionality in order to query RDF schemas. More precisely, we make the following contributions:

- In Section 2, we introduce a graph data model capturing RDF schema constructs [?, ?]. The originality of our model lies on the distinction between classes and relationship types (e.g. properties) in the style of ODMG [?, ?], as well as in the introduction of a graph instantiation mechanism, inspired by GRAM [?].
- In Section 3, we present the RQL language for querying semistructured RDF descriptions and schemas. RQL adopts the syntax and functional approach of OQL [?] while it features generalized path expressions in the style of POQL [?]. The novelty of RQL lies in its ability to query complex semistructured (meta)data and schema graphs using - in a transparent way - taxonomies of labels.
- In Section 4 we illustrate how we can take benefit from schema information in order to validate and efficiently store RDF descriptions in a DBMS. Furthermore, we sketch ideas on efficient schema indexing techniques for RQL query optimization.

2 Towards a Formal Model for RDF

In this section, we briefly recall the main modeling primitives proposed in the RDF Model & Syntax and Schema specifications [?, ?] and introduce our graph model (for more details see [?]).

RDF schemas are used to declare *classes* and *property-types*, typically authored for a specific community or domain. The upper part of Figure 1 illustrates such a schema for a cultural application. The scope of the declarations is determined by the *namespace* of the schema, e.g., $ns1$ (http://www.culture.gr/schema.rdf). Classes and property types are uniquely identified by prefixing their names with their schema namespace, as for example, $ns1$`#Artist` or $ns1$`#creates`. To simplify our presentation, we hereforth omit the namespace prefixes and denote by C the set of class names and by P the set of property types defined in a schema. Moreover, classes can be organized into a taxonomy through *simple* or *multiple* specialization. The root of this hierarchy, is a built-in class called `Resource`. For instance, `Painter` and `Painting` are subclasses of `Artist` and `Artifact` respectively, both specializing `Resource`. RDF classes do not impose any structure to their objects and class hierarchies simply carry inclusion semantics.

RDF property types serve to represent *attributes* (or *characteristics*) of resources as well as *relationships* (or *roles*) between resources. For example, *creates* defines a relationship between the resource classes `Artist` (its domain) and `Artifact` (its range) while *fname* is an attribute of `Artist` with type `Literal`[2] (i.e. range). As we can see in Figure 1, property types may

[2]As RDF literals we can have any primitive datatype defined in XML as well as XML markup which is not further interpreted by an RDF processor.

also be refined: *paints* is a specialization of *creates*, with its domain and range restricted to the classes `Painter` and `Painting`, respectively. We denote by $H = (N, \prec)$, a hierarchy of classes and property types, where $N = C \cup P$. H is *well-formed* if $\prec$ is a smallest partial ordering such that :

- if $c \in C$ then $c \prec Resource$ (i.e., the root of the class hierarchy).
- if $p_1, p_2 \in P$ and $p_1 \prec p_2$ then $domain(p_1) \prec domain(p_2)$ and $range(p_1) \prec range(p_2)$.

Besides literal and property types, RDF also supports *container* types, i.e., `Bag`, `Sequence` or `Alternative`. Members of containers are identified by a unique integer index label i, while no restriction is made on their types (i.e., may have heterogeneous member types). RDF classes and container types correspond to schema graph nodes whereas property types correspond to edges.

Definition 1 *An RDF schema is a directed labeled graph* $RS = (V_S, E_S, \psi, \lambda, H)$ *where,* V_S *is the set of nodes and* E_S *is the set of edges,* $H = (N, \prec)$ *is a well-formed hierarchy of classes and property types (including* $Bag, Seq, Alt, Literal$*),* λ *is a labeling function* $\lambda : V_S \cup E_S \rightarrow N$*, and* ψ *is an incidence function* $\psi : E_S \rightarrow V_S \times V_S$*, capturing the* domain *and* range *of properties.*

In RDF, *Resources* are described through a collection of *Statements* committing to a schema (see lower part of Figure 1). As a resource we may consider anything identifiable by an URI: it may be a Web page (e.g., http://www.museum.gr/picasso.htm), a fragment of a Web page (e.g., http://www.museum.gr/artstyles.xml#cubism) or an entire Web site (e.g., http://www.museum.gr). In the sequel, we denote by O the set of resource identifiers composed by a *namespace* and a file name or anchor id (e.g., &$ns2$#picasso, &$ns3$#cubism). A non-disjoint population function π assigns to each class c in C a set of object identifiers $\pi(c)$, such that: $\cup\{\pi(c') \mid c' \prec c\} \subseteq \pi(c)$.

A specific resource together with a named property and its value form an RDF statement, represented by an ordered pair $< v_1, v_2 >$, where v_1 is its subject and v_2 is its object. The subject (e.g., &$ns2$#picasso) and object (e.g., "Pablo") should be of a type compatible (under class specialization) with the domain and range of the used predicate (e.g., *fname*). Figure 1 shows that RDF properties can be *multi-valued* (e.g., two *paints* properties for &$ns2$#picasso), *optional* (e.g., there is no *fname* property for &$ns2$#rodin) and they can be inherited (e.g., the *creates* property of &$ns2$#rodin). Finally, resources can be *multiply classified* under several classes (e.g., &$ns2$#rodin is a `Painter` and a `Sculptor`). An RDF statement is simply an edge labeled with a property type, whereas an RDF description introduces a semistructured data graph. The semantics of edge and node labels in this graph is given by one or more associated RDF schemas.

Definition 2 *Given a population function* π*, an interpretation function is defined as follows:*

- *for a class* $c \in C$, $[\![c]\!] = \pi(c)$ *(note that* $[\![Resource]\!] = O$*),*
- *for a property type* $p \in P$, $[\![p]\!] = \{< v_1, v_2 >\mid v_1 \in [\![domain(p)]\!], v_2 \in [\![range(p)]\!]\} \cup \bigcup_{p' \prec p} [\![p']\!]$,
- *for a container type* $[\![Bag|Seq|Alt]\!] = \{1 : v_1, \ldots n : v_n\}$ *where* $v_1, \ldots v_n$ *are values in* O*.*

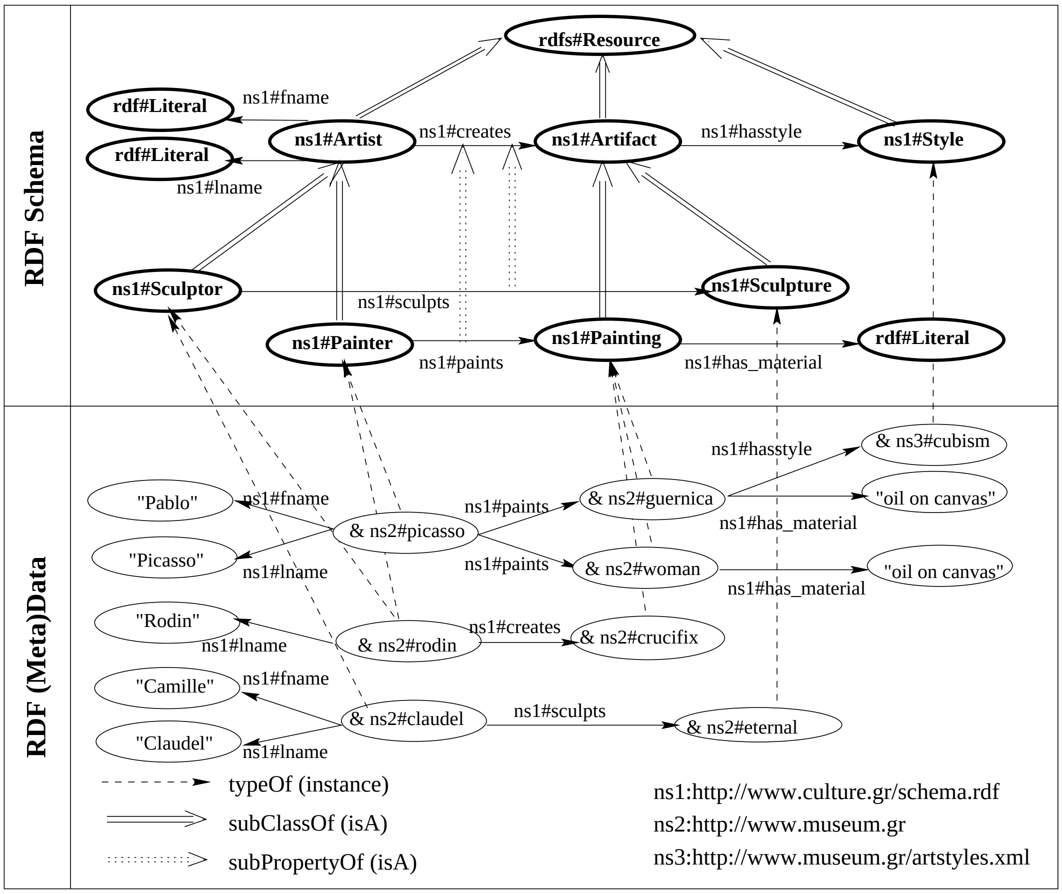

Figure 1: An example of semistructured RDF data and schemas

Definition 3 *An RDF description, instance of a schema RS, is a directed labeled graph* $RD = (V_D, E_D, \psi, \nu, \tau, O \cup L)$*, where:* V_D *is a set of nodes and* E_D *is a set of edges in an RDF data graph,* ψ *is the incidence function* $\psi : E_D \to V_D \times V_D$*,* ν *is a value function* $\nu : V_D \to O \cup L$ *and* τ *is a labeling function* $\tau : V_D \cup E_D \to N$ *which satisfies the following :*

- *for each node* v *in* V_D*,* $\tau(v)$ *is a set of names* $n \in C \cup \{Literal, Bag, Seq, Alt\}$ *where* $\nu(v) \in [\![n]\!]$*;*
- *for each edge* ϵ *from a node* v *to a node* v' *in* E_D*,* $\tau(\epsilon)$ *is a property type name* $p \in P \cup \{1, 2, \dots\}$*, such that* $\nu(v) \in [\![domain(p)]\!]$ *and* $\nu(v') \in [\![range(p)]\!]$*; additionally, if* $p \in \{1, 2, \dots\}$*,* v *should be of a container type:* $(Bag|Seq|Alt) \in \tau(v)$*.*

It should be stressed that our RDF graph model roughly corresponds to a finite, many-sorted relational structure. In fact, besides literal values and resource identifiers, the model relies on relations for class or property extents and containers. Note that resource URIs and names of class or property types may also be considered as values (i.e., strings), denoted as **val**. Then, an RDF data graph can be viewed as an instance of the following schema (with unnamed tuples):

```
cls(val)   prop(val, val)   cont(val, val, val)
```

Here `cls`, `prop` and `cont` correspond to specific schema classes, property types and to the *Bag*, *Seq*, *Alt* container types, respectively. Then `prop`($r1$,$r2$) indicates that $r1$,$r2$ are resource URIs connected through an edge labeled `prop`, while `cont`($s1$,1,$r2$) indicates that the first member of container value $s1$ is the resource $r2$. RDF schema vocabularies can also be represented using the relations `Class` and `Property` as well as two additional relations capturing the partial ordering ($\prec$) of classes and property types.

3 The RQL Query Language

In this section, we present the language *RQL* which allows us to query semistructured RDF descriptions using taxonomies of node and edge labels defined in an RDF schema. The following examples depict the use of *generalized path expressions* with variables on both kinds of labels.

Q1: *Find the resources created by a Painter, which have material "oil on canvas".*

```
select  Y
from    {X:$C}creates{Y}.has_material{Z}
where   $C = Painter, Z = "oil on canvas"
```

Y
&guernica
&woman

Q1 essentially implies a navigation through the structure of descriptions (i.e., through properties) and a filtering on both RDF data and schema information (e.g., classes). Data variables, like Y and Z are range-restricted to the *target* and *source* (considered as position indices) values respectively of the *creates* and *has_material* extents (see Section 2). Schema variables, prefixed with the symbol $, are range restricted to the meta-collections `Class` and `Property`. In **Q1**, $\$C$ denotes a class name variable, which is valuated to the domain (e.g., `Artist`) of the property *creates* and its subclasses (e.g., `Painter` and `Sculptor`). Then, the first condition in the `where` clause (i.e., equality of class names) restricts $\$C$ to `Painter`. The expression "$X : \$C$" (similar to a cast) restricts the *source* values of the *creates* extent only to the `Painter` instances, as for example, `&`$ns2$`#rodin` and `&`$ns2$`#picasso`. Note that if the class name in the `where` clause is not a valid subclass of the domain of *creates*, the query will return an empty answer. Moreover, the composition of paths, through the "`.`" operator in the `from` clause, implies a join between the extents of *creates* and *has_material* on their *target* and *source* values respectively. This way, *RQL* captures the existential semantics of navigation in semistructured data graphs: there exist two "*paints*" properties for `&`$ns2$`#picasso` while there is no "*has_material*" property for `&`$ns2$`#crucifix`, created by `&`$ns2$`#rodin` (declared also as a `Painter`). More formally, **Q1** is interpreted as:

$$\{v_2| \quad c_1 \in C, c_1 \prec domain(creates), v_1 \in [\![c_1]\!], < v_1, v_2 > \in [\![creates]\!], \\ < v_2, v_3 > \in [\![has\_material]\!],\ c_1 \ = \ Painter \ and \ v_3 \ = \ "oil \ on \ canvas"\}$$

RQL can also be used to query RDF schemas, regardless of any underlying instances. The main motivation for this is the use of *RQL* as a high-level language to implement schema browsing. This is justified by several reasons: a) in real applications RDF schemas may be very large, and therefore they cannot be manipulated in main memory [**?**]; b) due to class refinement, RDF schemas carry information about the labels of nodes and edges which is only implicitly stated in the schema graph (e.g., by inheritance of properties). Consider, for instance, the following query computing all the outgoing edges of a specific node (or nodes) in the schema graph:

Q2: *Find all the property types and their corresponding range, which can be used on a resource of type Painter or any of its subclasses.*

```
select  $P, $Y
from    {$X}$P{$Y}
where   $X <= Painter
```

$\$P$	$\$Y$
creates	Artifact
creates	Painting
paints	Painting

The formal interpretation of **Q2** is:

$$\{< p, c_2 > | \exists p \in P, c_1, c_2 \in C, c_1 \prec domain(p), c_2 \prec range(p), c_1 \prec Painter\}$$

Some of these edges are explicitly declared in the schema (e.g. *paints*) while others are inferred from the class hierarchy (e.g. *creates*). The same is true for the target nodes of the retrieved properties (e.g., `Painting` and `Artifact`). It should be stressed that due to multiple classification of nodes (e.g., &$ns2$#**rodin**), we can query paths in a data graph (e.g., in **Q1**) that are not included in the result of the corresponding schema queries (e.g., **Q2**). Still, the ability of *RQL* GPEs to combine filtering conditions on both graph data and schema, permits the querying of properties emanating from resources only, according to a specific class hierarchy (e.g., view the properties of &$ns2$#**rodin** only as a `Painter` and not as a `Sculptor`).

As a last example, we illustrate how *RQL* can be used to express the *AboutEachPrefix* retrieval function of RDF [**?**], returning both schema and data information.

Q3: *Tell me everything you know about the resources of the site "www.museum.gr".*

```
select  X, $Z, $P, Y
from    {X:$Z}$P{Y:$W}
where   Y like "*www.museum.gr*"
        or X like "*www.museum.gr*"
```

X	$\$Z$	$\$P$	Y	$\$W$
&rodin	Painter	creates	&crucifix	Painting
&rodin	Sculptor	creates	&crucifix	Painting
&picasso	Painter	paints	&guernica	Painting
&claudel	Sculptor	sculpts	&eternal	Sculpture
&picasso	Painter	fname	"Pablo"	Literal
&claudel	Sculptor	lname	"Claudel"	Literal
&guernica	Painting	hasstyle	&cubism	Style
...	...	...	...	...

Q3 will iterate over all property names ($\$P$), then for each property over its domain ($\$Z$) and range ($\$W$) classes and finally over the corresponding extents (X, Y). Finally, the result of *RQL* queries represented in this section in a tabular form (e.g., as $\neg$1NF relations) can be naturally captured by RDF Bag containers permitting heterogeneous member sorts (e.g., literals, URIs, sequences). Closure of *RQL* queries is ensured by supporting access operators for containers [**?**].

4 The RDF Metadata Management System

The metadata management system currently under development (see Figure 4) comprises three main components: the RDF validator and loader (**VRP**), the RDF description database (**DBMS**) and the query language interpreter (**RQL**).

RDF_Resource@4487	
URI	ns2#Picasso
rdf:type	ns1#Painter

RDF_Class@4455	
URI	ns1#Painter
rdf:type	rdfs:Class
rdfs:subClassOf	ns1#Artist

RDF_Property@5678	
URI	ns1#paints
rdf:type	rdf:Property
rdfs:subPropertyOf	ns1#creates
rdfs:domain	ns1#Painter
rdfs:range	ns1#Painting
link_list	(ns2#Picasso, ns2#Guernica)
	(ns2#Picasso, ns2#Woman)

Figure 2: Example objects in the VRP internal model

4.1 Parsing, Validation and Storage

The *Validating RDF Parser* (VRP) is a tool for analyzing, validating and processing RDF descriptions. Unlike existing RDF parsers (e.g. SiRPAC[3]), VRP[4] is based on standard compiler generator tools for Java, namely CUP/JFlex (similar to YACC/LEX). The stream-based parsing support of JFlex and the quick LALR grammar parsing of CUP ensure a good performance, when processing large volumes of RDF descriptions. The most distinctive feature of VRP is its ability to validate RDF descriptions against one or more schemas, as well as the schemas themselves.

The VRP validation module relies on an internal object model implemented in Java, separating RDF schemas from their instances. Instances of those schemas (i.e., resources) follow the graph model presented in section 2. More precisely, the VRP model consists of the following classes (see Figure 3): `Resource`, `RDF_Resource`, `RDF_Class`, `RDF_Property`, `RDF_Container` and `RDF_Statement`. Since, for RDF, everything is a resource, `Resource` is the root of the class hierarchy of the VRP internal model. Proper instances of this class represent the various resources (e.g., Web pages) in RDF descriptions which are identified by a URI (a hash map is used to transform string URIs to Java object ids). `RDF_Resource` is a direct subclass of `Resource`, representing resources with defined RDF/S properties (e.g., rdf:type, rdfs:label, rdfs:seeAlso). The other classes, `RDF_Class, RDF_Property, RDF_Container and RDF_Statement`,[5] are subclasses of RDF_Resource. The Java objects representing schema resources, i.e., classes and properties, are instances of the classes `RDF_Class` and `RDF_Property`. Figure 2 shows the objects that are created for the resources *ns*2`#Picasso`, *ns*1`#Painter` and *ns*1`#paints`, from the example of Figure 1.

This representation scheme, compared to the flat representation of triples produced by other RDF parsers, simplifies the manipulation of RDF metadata and schemas to a great extent. Firstly, the classification of resources (instances) in hierarchies makes semantics explicit.

[3]http://www.w3.org/RDF/Implementations/SiRPAC/

[4]http://www.ics.forth.gr/proj/isst/RDF

[5]The RDF_Statement class represents reified statements.

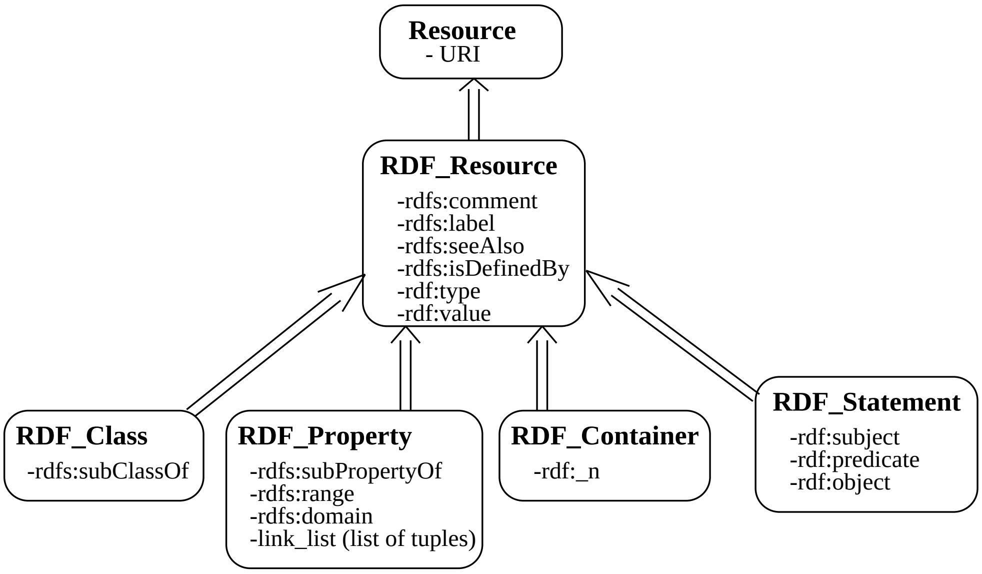

Figure 3: VRP internal object model

Moreover, the necessary information for loading such descriptions into a DB is straightforwardly represented in this model. Finally, by separating RDF Schemas from their instances, it allows easier manipulation of schema information, while verification of schema constraints can be performed more efficiently. This separation also facilitates a two-phase loading of schemas and their instances, as described below.

The Loader module APIs are based on the VRP internal model and comprise a number of primitive methods, which can be implemented for various DBMS technologies (e.g., relational, object). These primitive methods are defined as member functions of the classes of the VRP model, for storing the attribute values of the created objects. For example, the method `storetype()` is defined for the class `RDF_Resource`, in order to store type information of the objects. The primitive methods of each class are incorporated in a storage method defined in the respective class invoked during the loading process. The Loader takes advantage of the Java method-overriding mechanism, in order to store both RDF descriptions and schemas in a DBMS using a two-phase algorithm: During the first phase, RDF schema information (i.e., class and property descriptions) is loaded in the database, to create the corresponding storage schema (i.e., tables for RDBMS, classes for ODBMS). It should be stressed that the storage schema is a direct image of the associated RDFS schemas as presented in section 2. During the second phase, this schema is used to populate the database with resource descriptions.

For example, Figure 4 illustrates the representation of RDF descriptions in a *relational DBMS*, using specific schema information. It consists of four tables capturing the class and property-type hierarchies defined in an RDF schema, namely `Class`, `Property`, `SubClass` and `SubProperty`. Then, for every new class or property loaded in the database, we create a new table to store its instances. This implementation conforms to our graph model and permits a uniform representation of both RDF descriptions and schemas, while capturing in a precise way the semantics of the latter.

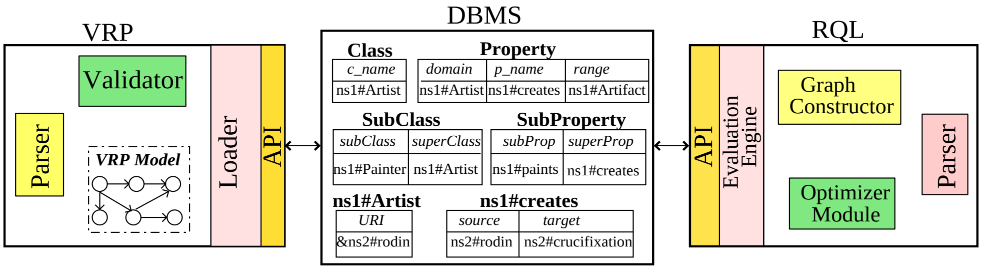

Figure 4: System architecture

4.2 Query Processing

The *RQL interpreter* consists of (a) the parser, analyzing the syntax of queries; (b) the graph constructor, reflecting the semantics of queries and (c) the evaluation engine, accessing RDF descriptions and schema information from the underlying database. As in the case of the loader, the RQL evaluation engine relies on high-level APIs that can be implemented as front-end access functions for various DBMS technologies. The development of the *RQL optimizer* is ongoing and will be mainly based on heuristic methods for query rewriting (join reordering, etc.), making use of realistic assumptions about the queried extents and exploiting possible index structures. In particular, we plan to implement indices for RDF schema classes (or property-type) hierarchies (see *Subclass* and *SubProperty* relations) in order to handle efficiently recursive access to all subclasses (or subproperties) of a given class (or property) as in the case of query **Q2** (see Section 3).

In applications where the RDF schema contains deep and voluminous classification hierarchies, queries accessing subclasses or subproperties of a given class or property respectively, are extremely time consuming. As demonstrated in [?], in cultural applications a schema could consist of rather deep and broad taxonomies of concepts (*terms*) originating from application specific vocabularies. In [?] the authors demonstrate the creation of an *RDF* schema by integrating the rather shallow *ICOM/CIDOC Reference Model* [?] and the rich *Art & Architecture Thesaurus* [?]. The former is a conceptual schema defined by the International Council of Museums to describe cultural information, containing around 30 concepts and 60 roles. The latter is one of the largest thesauri in the area of western art and historical terminology containing around 28.000 terms. In the schema resulting from the integration of the above conceptual structures, ICOM/CIDOC concepts and AAT terms are modeled as *RDF classes*, the latter considered as *sub-classes* of the former, organised in monohierarchical inheritance taxonomies. Those simple inheritance hierarchies are rather deep and broad and queries that require access to the subtree of a given class or property are essentially traversal queries over the *SubClass* relation of Figure 4, and are rather costly. The idea is to transform such traversal queries into interval queries on a linear domain, that can be answered efficiently by standard DBMS index structures. To do this, node[6] names are replaced by *ids* for which a convenient total order exists. A simple encoding to provide those *ids* is the following: assuming that the *fan-out*, i.e., the number of immediate descendants of a node in the schema, is less than 9 and that *max* is the maximum number of immediate descendants of a node, then each node is ranked with a number i, $i \leq max$ (see Figure 5(a)). Then to each node k in this linear domain, we associate an *id*, whose length is d, if the node is at depth d in the

[6]From this point on, we will be refering to a class or property as a node.

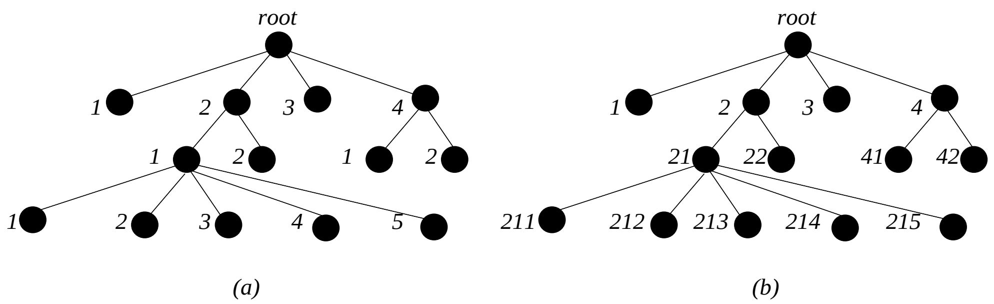

Figure 5: Linear Encoding of classes

hierarchy. The j^{th} character of id is m, if the j^{th} node is the mth sibling of k (Figure 5(b)). It is evident that if $id(k)$ is the id of node k, then all ids in the tree whose root is k have ids larger than k. As illustrated in Figure 5(b), if we consider the node with label 21, then its immediate descendants have ids $211, 212, ..$ and its right sibling has label $22 > 21$. We denote the right sibling of a node n by $next(n)$. Using this encoding, all descendants of a node (class or property) n, which is in fact a typical query, are obtained by executing the interval query $id(X)\ \ in\ [id(n), id(next(n))]$. For example a query on node n with $id = 2$, returns all nodes whose labels are between $id(n)(2)$ and $id(next(n))(3)$. So the previous query $Q2$ will be transformed into $id(X)\ \ in\ [id(Painter), nextid(Painter)]$. We must mention here that in order to be able to transform traversal queries into simple interval queries, we need to store for each term its id, and the id of its next right sibling. We use regular B-tree based optimization techniques to answer such queries. This approach has been presented more thoroughly in [**?**], where the authors demonstrate how such a linear representation of nodes can be used to efficiently answer queries that involve traversal of deep classification hierarchies (e.g., thesauri).

5 Conclusions and Future Work

This paper puts forth the idea that declarative query languages for metadata, like RQL, open new perspectives in the effective and efficient support of WWW applications. RQL can be used as high-level language (opposed to existing APIs [**?**]) to access various RDF metadata repositories, by exploiting its ability to uniformly query (meta)data and schema vocabularies and to handle incomplete information. Especially RQL can exploit transparently taxonomies of classes and types in order to facilitate querying of complex semistructured data using only few abstract labels. The paper also presented an architecture for metadata management comprising efficient mechanisms for parsing and validating RDF descriptions, loading into a DBMS and RQL query processing and optimization.

Current research and development efforts focus on desining appropriate access path selection mechanisms and heuristic methods for query rewriting and optimization. Appropriate index structures for reducing the cost of recursive querying of deep hierarchies need to be devised as well. Specifically, an implementation of hieararchy linearization is under way, exploring alternative node encodings. The performance of the system will be assessed using benchmarks for relational and object-oriented DBMS platforms.

References