Querying Community Web Portals*

Gregory Karvounarakis Vassilis Christophides Dimitris Plexousakis Sofia Alexaki
Institute of Computer Science,
FORTH, Vassilika Vouton, P.O.Box 1385, GR 711 10, Heraklion, Greece
and
Department of Computer Science,
University of Crete, GR 71409, Heraklion, Greece
{gregkar, christop, dp, alexaki}@ics.forth.gr

Abstract

A new generation of information systems such as organizational memories, vertical aggregators, infomediaries, etc. is emerging nowadays. Such systems, termed Community Web Portals, intend to support specific communities of interest (e.g., enterprise, professional, trading) on corporate intranets or the Web. More precisely, Portal Catalogs, organize and describe various information resources (e.g., sites, documents, data) for diverse target audiences (corporate, inter-enterprise, e-marketplace, etc.), in a multitude of ways, which are far more flexible and complex than those provided by standard (relational or object) databases. Yet, in commercial software for deploying Community Portals, querying is still limited to full-text (or attribute-value) retrieval and more advanced information-seeking needs implies navigational access. Furthermore, recent Web standards for describing resources are completely ignored.

In this paper, we propose a declarative language suitable for querying Portal Catalogs created according to the Resource Description Framework (RDF) W3C standard. Our language, called RQL, relies on a formal graph model, that captures the RDF modeling primitives and permits the interpretation of heterogeneous descriptions by means of one or more schemas. In this context, RQL adapts the functionality of semistructured query languages to the peculiarities of RDF but also extends this functionality in order to uniformly query both resource descriptions and related schemas. Then, RQL enables to query Portal Catalogs holding multi-purpose descriptions of community resources while preserving a conceptually unified view of the Catalog for (sub-)communities employing different RDF schemas. RQL is actually used by several applications aiming at building, accessing and personalizing Community Web Portals.

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

1 Introduction

Information systems such as organizational memories, vertical aggregators, infomediaries, etc. are expected to play a central role in the 21st-century economy by enabling the development and maintenance of specific communities of interest (e.g., enterprise, professional, trading) on corporate intranets or the Web [26]. Such *Community Web Portals* essentially provide the means to select, classify and access, in a semantically meaningful and ubiquitous way various information resources (e.g., sites, documents, data) for diverse target audiences (corporate, inter-enterprise, e-marketplace, etc.). The core Portal component is a *Catalog* holding descriptions, i.e., *metadata*, about the resources available to the community members. In order to effectively disseminate community knowledge, Portal Catalogs organize and gather information in a *multitude of ways*, which are far more flexible and complex than those provided by standard (relational or object) databases [45, 10, 34]. Yet, in commercial software for deploying Community Portals (e.g., Epicentric, Plumtree, Glyphica[1]), querying is still limited to full-text (or attribute-value) retrieval and more advanced information-seeking needs implies navigational access. Furthermore, recent Web standards for describing resources (see the W3C Metadata Activity[2]) are completely ignored.

More precisely, the Resource Description Framework (RDF) standard [33, 9] proposed by W3C intends to facilitate the creation and exchange of resource descriptions between Community Webs. Due to its flexible model, many content providers (e.g., ABCNews, CNN, Time Inc.) and Web Portals (e.g., Open Directory, CNET, XMLTree[3]) or browsers (e.g., Netscape 6.0, W3C Amaya) already adopt RDF [33]. As a matter of fact, by considering community information as a collection of resources identified by URIs and by modeling resource descriptions using named properties, RDF enables the provision of various kinds of metadata (for administration, recommendation, content rating, site maps, push channels, etc.) about resources of quite diverse nature (ranging from PDF or Word documents, e-mail or audio/video files to HTML pages or XML data). Moreover, to interpret resource descriptions within or across communities, RDF allows the definition of schemas [9] which (a) do not impose a strict typing on the descriptions (e.g., a resource may be liberally decsribed using properties which are loosely-coupled with entity classes); (b) permit heterogeneous descriptions of the same resources for different purposes (e.g., by classifying resources to multiple classes which are not necessarily related by subclass relationships); (c) can be easily extended to meet the description needs of specific (sub-)communities (e.g., through specialization of both entity classes and properties).

Strictly speaking, in most Community Web Portals, resources may have quite heterogeneous descriptions, that can be easily and naturally represented in RDF as *directed labeled graphs* where nodes are called *resources* (or *literals*) and edges are called *properties*. Hence, RDF schemas essentially define vocabularies of labels (for graph nodes and edges) that can be used to describe and

[1] www.epicentric.com www.plumtree.com www.glyphica.com
[2] www.w3.org/Metadata
[3] www.dmoz.org home.cnet.com www.xmltree.com

query resources in different application contexts. Both kinds of labels can be organized into taxonomies carrying inclusion semantics (i.e., class or property subsumption). These modeling primitives are reminiscent to well-known knowledge representation languages (e.g., TELOS [41, 44]) and they are quite useful for supporting multi-purpose descriptions of community resources while preserving the autonomy of descriptions for different (sub-)communities. Note that similar needs are also exhibited in other net-based applications such as Superimposed Information Systems [22, 36] and LDAP Directory Services [30, 6]. In a nutshell, the growing number of available information resources and the proliferation of description services in communities, lead nowadays to large volumes of RDF metadata (e.g., the Open Directory Portal of Netscape comprises around 100M of Subject Topics and 700M of indexed URIs). It becomes evident that browsing such large *description bases* is a quite cumbersome and time-consuming task. Unfortunately, this is the only support provided by existing RDF-enabled systems [46] relying on low-level APIs and file-based implementations. The question that naturally arises is to what extent can DBMS technology be used to support declarative access and secondary storage management for RDF description bases.

Clearly, standard (relational or object) databases [4] are too rigid for capturing the peculiarities of RDF descriptions and schemas (e.g., RDF classes do not define relation or object types and their instances may have quite different associated properties). On the other hand, most semistructured formalisms, such as OEM [43, 42] or UnQL [11], are totally schemaless (allowing arbitrary labels on edges or nodes but not both). Moreover, semistructured systems offering typing features (e.g., pattern instantiation) like YAT [17, 18], cannot exploit the RDF class (or property) hierarchies. Finally, RDF schemas have substantial differences from XML DTDs [8] or the more recent XML Schema proposal [48, 37] (e.g., extension and refinement of XML element content models is purely syntactic while XML elements may have at most one content model). As a consequence, query languages proposed for semistructured or XML data (e.g., LOREL [3], StruQL [25], XML-QL [24], XML-GL [14], Quilt [20]) fail to interpret the semantics of RDF node or edge labels. The same is true for the languages proposed to query the schema of (relational or object) databases (e.g., SchemaSQL [32], XSQL [31], Noodle [40]) as well as for logic-based (F-Logic or Datalog) RDF query languages (e.g., SiLRI [21], Metalog [38]).

Motivated by the above issues, we propose a new query language for RDF descriptions and schemas. Our language, called *RQL*, relies on a formal graph model that captures the RDF modeling primitives (i.e., labels on both graph nodes and edges, taxonomies of labels) and permits the interpretation of heterogeneous resource descriptions by means of one or more schemas. In this context, *RQL* adapts the functionality of semistructured query languages to the peculiarities of RDF but also extends this functionality in order to *uniformly query* both RDF descriptions and schemas. As a matter of fact, *RQL* is a generic tool that can be used by several applications aiming at building, accessing and personalizing Community Web Portals. An example of a Portal created for a Cultural Community is given in Section 2. Then, we make the following contributions:

- We introduce in Section 3 the first formal data model for *description bases* created according to the RDF Model & Syntax and Schema specifications [33, 9]. The main challenge for this model is the representation of properties as *self-existent* individuals, as well as the introduction of a graph instantiation mechanism permitting multiple classification of resources (i.e., nodes). Compared to the current status of the W3C RDFS recommendation, our graph model provides a richer type system including several basic types as well as union types.
- We propose in Section 4 a declarative language, called RQL, to query both RDF descriptions and related schemas. RQL is a typed language following a functional approach (a la OQL [13]) and supports *generalized path expressions* featuring variables on both labels for nodes (i.e., classes) and edges (i.e., properties). The novelty of RQL lies in its ability to smoothly switch between schema and data querying while exploiting - in a transparent way - the taxonomies of labels and multiple classification of resources. The functionality and formal interpretation of RQL is given for several classes of useful queries required by Community Web Portals. To the best of our knowledge, RQL is the first language offering this functionality.
- We describe in Section 5 the implementation of RQL on top of an object-relational DBMS (ORDBMS). We illustrate how RDF descriptions can be represented in an ORDBMS taking into account the related schemas and sketch how RQL queries are translated into SQL3. More precisely, we focus on the algebraic rewriting performed by the RQL optimizer to push the maximum of path expressions evaluation (involving both schema and data querying) to the underlying ORDBMS.

Finally, Section 6 presents our conclusions and draws directions for further research.

2 Example of a Cultural Community Web Portal

In this section, we briefly recall the main modeling primitives proposed in the Resource Description Framework (RDF) Model & Syntax and Schema (RDFS) specifications [33, 9]. Our presentation relies on an example of a Portal Catalog created for a Cultural Community. To build this Catalog we need to describe cultural resources (e.g., Museum Web sites, Web pages with exhibited artifacts) both from a Portal administrator and a museum specialist perspective. The former is essentially interested in management metadata (e.g., mime-types, file sizes, modification dates) of resources, whereas the latter needs to focus more on their semantic description using notions such as Artist, Artifact, Museum and their possible relationships. These semantic descriptions can be constructed using existing ontologies (e.g., the International Council of Museums CIDOC Reference Conceptual Model[4]) or vocabularies (e.g., the Open Directory Topics hierachy[5]) employed by communities[6] and cannot always be extracted automatically from resource content or hyperlinks.

[4] www.ics.forth.gr/proj/isst/Activities/CIS/cidoc

[5] www.dmoz.org

[6] Note that the complexity of semantic descriptions depends on the diversity of resources and the breadth of community domains of discourse.

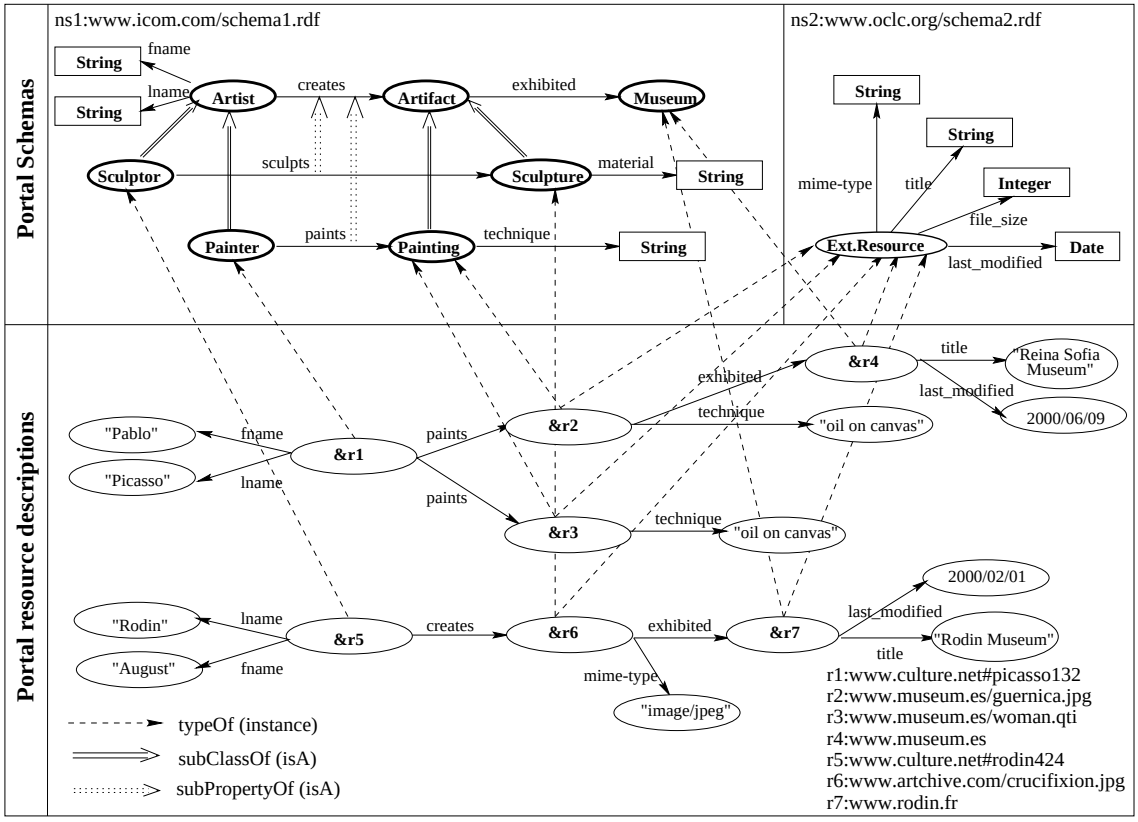

Figure 1: An example of RDF resource descriptions for a Cultural Portal

In the lower part of Figure 1, we can see the descriptions created for two Museum Web sites (resources `&r4` and `&r7`) and three images of artifacts available on the Web (resources `&r2`, `&r3` and `&r6`). We hereforth use the notation `&` to denote the involved resource URIs (i.e., identity). For example, `&r4` is first described as an `ExtResource` having two properties: `title` with value the string "Reina Sophia Museum" and `last_modified` with value the date 2000/06/09. Then, `&r4` is also classified under `Museum`, in order to capture its semantic relationships with other Web resources such as artifact images. For instance, we can state that `&r2` is of type `Painting` and has a property `exhibited` with value the resource `&r4` and a property `technique` with string value "oil on canvas". Resource `&r2` as well as `&r3` and `&r6` are also *multiply classified* under `ExtResource`. Finally, in order to interrelate artifact resources, some intermediate resources for artists (i.e., which are not on the Web) need to be generated, as for instance, `&r1` and `&r5`. More precisely, `&r1` is a resource of type `Painter` and its URI is given internally by the Portal description base. Associated with `&r1` are: a) two `paints` properties with values the resources `&r2` and `&r3`; and b) an `fname` property with value "Pablo" and an `lname` property with value "Picasso". Hence, diverse descriptions of the same Web resources (e.g., `&r2` as `ExtResource` and `Museum`) are easily and naturally represented in RDF as *directed labeled graphs*. The labels for graph nodes (i.e., classes or litteral types) and edges (i.e., properties) that can be employed to describe and query resources are defined in RDF schemas.

In the upper part of Figure 1, we can see two such schemas: the first intended for museum spe-

cialists while the second for Portal administrators. The scope of the declarations is determined by the corresponding *namespace* definition of each schema, e.g., **ns1** (`www.icom.com/schema1.rdf`) and **ns2** (`www.oclc.com/schema2.rdf`). For simplicity, we will hereforth omit the namespaces prefixing class and property names. In the former schema, the property `creates`, has been defined with domain the class `Artist` and range the class `Artifact`. Note that properties serve to represent *attributes* (or *characteristics*) of resources as well as *relationships* (or *roles*) between resources and they have unique names. Furthermore, both classes and properties can be organized into taxonomies carrying inclusion semantics (multiple specialization is also supported). For example, the class `Painter` is a subclass of `Artist` while the property `paints` (or `sculpts`) refines `creates`. In a nutshell, *RDF properties are decoupled from class definitions* and are by default *unordered* (e.g., there is no order between the properties `fname` and `lname`), *optional* (e.g., the property `material` is not used), *multi-valued* (e.g., we have two `paints` properties), and they can be *inherited* (e.g., `creates`).

A specific resource (i.e., node) together with a named property (i.e., edge) and its value (i.e., node) form a *statement* in the RDF jargon. Each statement is represented by a *triple* having a *subject* (e.g., `&r1`), a *predicate* (e.g., `fname`), and an *object* (e.g., "Pablo"). The subject and object should be of a type compatible (under class specialization) with the domain and range of the predicate (e.g., `&r1` is of type `Painter`). In the rest of the paper, the term *description base* will be used to denote a collection of RDF statements. Although not illustrated in Figure 1, RDF also supports structured values called *containers* (i.e., bag, sequence) for grouping statements, as well as, higher-order statements (i.e., *reification*[7]). Finally, both RDF graph schemas and descriptions can be serialized in XML (see Appendix A for the XML syntax of our example) using various forests of XML trees (i.e., there is not root XML node).

The flexibility of the RDF/S data model allows us to refine Portal schemas and/or enrich descriptions at any time while it ensures the autonomy of description bases for different (sub-) communities. In this context, a Portal may comprise multi-purpose descriptions of its resources while preserving a conceptually unified view of its description base (i.e., its Catalog) through one or the union of all related schemas. It becomes clear that RDF modeling primitives are substantially different from those defined in object or relational database models [4]. More precisely:

- *Classes do not define object or relation types*: an instance of a class is just a resource URI without any value/state (e.g., the URI `&r2` is an instance of `Painting` regaless of any property associated to it);
- *Resources may belong to different classes* not necessarily pairwise related by specialization: the instances of a class may have associated quite different properties while there is no other class on which the union of these properties is defined (e.g., the different properties of `&r2` and `&r4` which both are instances of `ExtResource`);

[7] We will not treat reification in this paper.

- *Properties may also be refined* by respecting a minimal set of constraints i.e., domain and range compatibilities (e.g., the property `creates`).

In addition, less rigid models such as proposed for semistructured or XML databases [1] also fail to capture the semantics of RDF description bases. Either they are completely schemaless, or the structure of an RDF description cannot be represented by element content models (i.e., types) foreseen in XML DTDs [8] or Schemas [48, 37]: due to multiple classification, resources may have quite irregular structures modeled only through an exception mechanism a la SGML [29]. Still we want to take benefit from three decades of research in high-level database query languages in order to access Portal description bases. Then, Portal applications have to specify only which resources need to be accessed, leaving the task of determining how to efficiently access them to the Portal query engine. Our data model and query language for Community Web Portals has been designed in this respect and they will be presented in the sequel.

3 The RDF Data Model

In this section we present a graph-based model that caters to the peculiarities of the RDF Model & Syntax and Schema (RDFS) specifications [33, 9] (represented respectively by the namespaces `rdf` and `rdfs`). The main objective of this formal model is to represent properties as *self-existent* individuals, and introduce a richer, but still flexible, type system for RDF description bases.

We assume the existence of the following countably infinite and pairwise disjoint sets of symbols: $\mathcal{C}$ of *Class names*, $\mathcal{P}$ of *Property names*, $\mathcal{U}$ of *Resource URIs* as well as a set $\mathcal{L}$ of *Literal type names* like *string*, *integer*, *date*, etc. Each literal type $t \in \mathcal{L}$ has an associated domain, denoted $dom(t)$ and $dom(\mathcal{L})$ denotes $\bigcup_{t \in \mathcal{L}} dom(t)$ (i.e., the `rdfs:Literal` declaration). Without loss of generality, we assume that the sets $\mathcal{C}$ and $\mathcal{P}$ are extended to include as elements the class name `Class` and the property name `Property` respectively. The former captures the root of a class hierarchy (i.e., the `rdfs:Class` declaration) while the latter the root of a property hierarchy (i.e., the `rdf:Property` declaration) defined in RDF/S [33, 9]. Additionally, $\mathcal{P}$ also contains integer labels ($\{1, 2, \ldots\}$) used as property names (i.e., the `rdfs:ContainerMembershipProperties` declaration) by the members of container values (i.e., the `rdfs:Bag`, `rdfs:Sequence` declarations).

Each RDF schema uses a finite set of class names $C \subseteq \mathcal{C}$ and property names $P \subseteq \mathcal{P}$. Property types are then defined using class names or literal types so that: for each $p \in P$ $domain(p) \in C$ and $range(p) \in C \cup \mathcal{L}$. We denote by $H = (N, \prec)$ a hierarchy of class and property names, where $N = C \cup P$. H is *well-formed* if $\prec$ is a smallest partial ordering such that:

- if $c \in C$ then $c \prec Class$;
- if $p \in P$ then $p \prec Property$;
- if $p_1, p_2 \in P$ and $p_1 \prec p_2$, then $domain(p_1) \preceq domain(p_2)$ and $range(p_1) \preceq range(p_2)$[8].

[8] The symbol $\preceq$ extends $\prec$ with equality.

In this context, RDF data can be atomic values (e.g., strings), resource URIs, and container values holding query results, namely *rdf:Bag* (i.e., multi-sets) and `rdf:Sequence` (i.e., lists). The notion of sets in RDF is less important. More precisely, the main types foreseen by our model are:

$$\tau = \tau_L \mid \tau_U \mid \{\tau\} \mid [\tau] \mid (1 : \tau + 2 : \tau + \ldots + n : \tau)$$

where τ_L is a literal type in $\mathcal{L}$, $\{.\}$ is the Bag type, $[.]$ is the Sequence type, $(.)$ is the Alternative type, and τ_U is the type for resource URIs (in Section 4, we will see that our query language treats URIs, i.e., identifiers, as simple strings). Alternatives capture the semantics of union (or variant) types [12], and they are also ordered (i.e., integer labels play the role of union member markers). Since there exists a predefined ordering of labels for sequences and alternatives, labels can be omitted (for bags, labels are meaningless). Furthermore, all types are mutually exclusive (e.g., a literal value cannot also be a bag) and no subtyping relation is defined in RDF/S. The set of all type names is denoted by T.

This type system offers all the arsenal we need to capture containers with both homogeneous and heterogeneous member types, as well as, to interpret RDF schema classes and properties. For instance, *unnamed ordered tuples* denoted by $[v_1, v_2, \ldots, v_n]$ (where v_i is of some type τ_i) can be defined as heterogeneous sequences[9] of type $[(\tau_1 + \tau_2 + \ldots + \tau_n)]$. Hence, RDF classes can be seen as unary relations of the type $\{\tau_U\}$ while properties as binary relations of type $\{[\tau_U, \tau_U]\}$ (for relationships) or $\{[\tau_U, \tau_L]\}$ (for attributes). As we will see in Section 4, RDF containers can be used to represent n-ary relations (e.g., as a bag of sequences). Finally, assignment of a finite set of URIs (of type τ_U) to each class name[10] is captured by a *population function* $\pi : C \rightarrow 2^U$. The set of all values forseen by our model is denoted by V.

Definition 1 *The interpretation function $[\![.]\!]$ is defined as follows:*

- *for literal types: $[\![\mathcal{L}]\!] = dom(\mathcal{L})$;*
- *for the Bag type, $[\![\{\tau\}]\!] = \{v_1, v_2, \ldots v_n\}$ where $v_1, v_2, \ldots v_n \in V$ are values of type τ;*
- *for the Seq type, $[\![[\tau]]\!] = [v_1, v_2, \ldots v_n]$ where $v_1, v_2, \ldots v_n \in V$ are values of type τ;*
- *for the Alt type $[\![(\tau_1 + \tau_2 + \ldots + \tau_n)]\!] = v_i$ where $v_i \in V$ $1 < i < n$ is a value of type τ_i;*
- *for each class $c \in C$, $[\![c]\!] = \pi(c) \cup \bigcup_{c' \prec c} [\![c']\!]$;*
- *for each property $p \in P$, $[\![p]\!] = \{[v_1, v_2] \mid v_1 \in [\![domain(p)]\!], v_2 \in [\![range(p)]\!]\} \cup \bigcup_{p' \prec p} [\![p']\!]$.*

In the sequel we will use the notation $\hat{}[\![.]\!]$ to distinguish between strict and extended interpretation of classes and properties.

Definition 2 *An RDF schema is a 5-tuple $RS = (V_S, E_S, \psi, \lambda, H)$, where: V_S is the set of nodes and E_S is the set of edges, H is a well-formed hierarchy of class and property names $H = (N, \prec)$, λ is a labeling function $\lambda : V_S \cup E_S \rightarrow N \cup T$, and ψ is an incidence function $\psi : E_S \rightarrow V_S \times V_S$.*

[9] Observe that, since tuples are ordered, for any two permutations $i_1, \ldots, i_n$ and $j_1, \ldots, j_n$ of $1, \ldots, n$, $[i_1 : v_1, \ldots, i_n : v_n]$ is distinct from $[j_1 : v_1, \ldots, j_n : v_n]$.

[10] Note that we consider here a non-disjoint object id assignment to classes due to multiple classification.

The *incidence function* captures the `rdfs:domain` and *rdfs:range* declarations of properties. Note that the incidence and labeling functions are *total* in $V_S \cup E_S$ and E_S respectively. This does not exclude the case of schema nodes which are not connected through an edge. Additionally, we impose a *unique name assumption* on the labels of RS nodes and edges.

Definition 3 *An RDF description base, instance of a schema RS, is a 5-tuple* $RD = (V_D, E_D, \psi, \nu, \lambda)$, *where:* V_D *is a set of nodes and* E_D *is a set of edges,* ψ *is the incidence function* $\psi : E_D \rightarrow V_D \times V_D$, ν *is a value function* $\nu : V_D \rightarrow V$, *and* λ *is a labeling function* $\lambda : V_D \cup E_D \rightarrow 2^{N \cup T}$ *which satisfies the following:*

- *for each node* v *in* V_D, λ *returns a set of names* $n \in C \cup T$ *where the value of* v *belongs to the interpretation of each* n: $\nu(v) \in [\![n]\!]$;
- *for each edge* ϵ *in* E_D *going from node* v *to node* v', λ *returns a property name* $p \in P$, *such that:*
  - *if* $p \in P \setminus \{1, 2, \ldots\}$, *the values of* v *and* v' *belong respectively to the interpretation of the domain and range of* p: $\nu(v) \in [\![domain(p)]\!]$, $\nu(v') \in [\![range(p)]\!]$;
  - *if* $p \in \{1, 2, \ldots\}$, *the values of* v *and* v' *belong respectively to the interpretation of a* $Bag|Seq|Alt$ *type and their corresponding member types:* $\nu(v) \in [\![Bag|Seq|Alt(\tau)]\!]$, $\nu(v') \in [\![\tau]\!]$.

Note that the *labeling function* captures the `rdf:type` declaration that associates each RDF node with one or more class names and several schemas can be used to label RDF description graphs.

To conclude this section, we note that our model captures the majority of RDF/S modeling primitives with the exception of (a) properties having *multiple domain* definitions, since it is always possible to prefix these properties with their domain class names; (b) property *reification* given that it is not expressible in RDFS. Finally, built-in property types such as `rdfs:seeAlso`, `rdfs:isDefinedBy`, `rdfs:comment`, `rdfs:label` can be easily represented in our model, but due to space limitations they are not considered in this paper.

4 The RDF Query Language: RQL

RQL is a typed query language relying on a functional approach (a la OQL [13]). It is defined by a set of basic queries and iterators which can be used to build new ones through functional composition of side-effect free functions (see Appendix B for the complete syntax and Appendix C for the typing system). Furthermore, *RQL* supports generalized path expressions, featuring variables on labels for both nodes (i.e., classes) and edges (i.e., properties). The novelty of *RQL* lies in its ability to smoothly combine schema and data path expressions while exploiting - in a transparent way - the taxonomies of classes and properties, as well as, the multiple classification of resources. As we will see in the sequel this functionality is required by several Community Web Portal applications (e.g., simple browsing, personalization, interactive querying, etc.).

To uniformly query nodes and edges either in RDF descriptions or in schemas, *RQL* blurs the distinction between schema labels (for classes, properties and types) and resource labels (i.e., URIs and literal values). In the rest of the paper we consider that both kinds of labels can be treated as strings and the interpretation of τ_U is extended as follows: $[\![\tau_U]\!] = T \cup C \cup P \cup U$. Abusing notation, we use τ_{UC} (τ_{UP}) to denote the type of class (property) names in schemas and τ_{UR} to denote only the URIs of resources in description bases.

4.1 Browsing Portals using RQL Basic Queries

The core *RQL* queries essentially provide the means to access RDF description bases with minimal knowledge of the employed schema(s). These queries can be used to implement a simple browsing interface for Community Web Portals. For instance, in Web Portals such as Netscape Open Directory[11], for each topic (i.e., class), one can navigate to its subtopics (i.e., subclasses) and eventually discover the resources which are directly classified under them. In this subsection we will see how the basic *RQL* queries (see Table 1 for formal definitions) can be used to generate such Portal interfaces, either off-line (i.e., by materializing the various query results in HTML/XML files) or online (by computing query answers on the fly). Up to now this access functionality is supported by existing RDF-enabled systems [46] using low-level APIs and file-based implementations.

To warmup readers, we start with queries which can find all the schema classes or properties used in a Portal Catalog:

```
Class

Property
```

`Class` and `Property` hold all the labels of nodes and edges that can be used in an RDF description base. In our example, these basic queries will return the URIs of the classes (of type τ_{UC}) and properties (of type τ_{UP}) illustrated in the upper part of Figure 1. Then, for a specific property we can find its definition by applying the corresponding `domain` (of type τ_{UC}) and `range` (of type τ_{UC} for attributes and τ_{UL} for relationships) functions. For instance, `domain(creates)` will return the class name *Artist*. To traverse the class/property hierarchies, *RQL* provides various functions such as `subClassOf` (for transitive subclasses) and `subClassOf^` (for direct subclasses). For example, the following query will return the class URIs *Painter* and *Sculptor*:

```
subClassOf^(Artist)
```

More generally, we can access any RDF collection by just writing its name. This is the case of RDF classes considered as unary relations:

```
Artist
```

This query will return the bag containing the URIs `www.culture.net#rodin424` and `www.culture.net#picasso132` since these resources belong to the extent of *Artist*. It should be stressed that, by default, we consider an extended class interpretation. This is motivated by the

[11] www.dmoz.org

Expression	Interpretation
`subClassOf(c)`	$\{c' \in C \mid c' \preceq c\}$
`subClassOf^(c)`	$\{c' \in C \mid c' \preceq c, \nexists c'' \in C, c'' \prec c \& c' \prec c''\}$
`p`	$[\![p]\!]$
`^p`	$[\![p]\!] - \bigcup_{p' \prec p}([\![p']\!])$
col_1 `intersect` col_2	$[\![col_1]\!] \cap [\![col_2]\!]$
`count(`col`)`	n, if $\exists i_1, \ldots, i_n, \exists v_{i_1}, \ldots, v_{i_n}$, $\forall j, i_1 \leq j \leq i_n \ (j, v_j) \in [\![col]\!]$
`element(`col`)`	$\{v \mid v \in \{v\}\}$
o `in` col	$\exists i, 1 \leq i \leq count(col), (i, o) \in [\![col]\!]$
$seq[n:m]$	$< v_n, \ldots, v_m >$ where $\exists j_1, \ldots, j_l, 1 \leq n \leq m \leq l$, $\exists v_1, \ldots, v_{(n-1)}, \ldots, v_{(m+1)}, \ldots, v_l, \forall i, 1 \leq i \leq l, (j_i, v_i) \in [\![seq]\!]$

Table 1: Formal Interpretation of the core RQL Queries

fact that class (or property) names can be simply viewed as *terms* of an RDF schema vocabulary, and RQL offers a term expansion mechanism similar to that of thesauri-based information retrieval systems [28]. In order to obtain the proper instances of a class (i.e., only the nodes labeled with the class URI), RQL provides the special operator ("`^`"). In our example, the result of `^Artist` will be the empty bag. Additionally, we can inspect the cardinality of class extents (or any other collection) using the `count` function.

It should be stressed that RQL considers as entry-points to an RDF description base not only the names of classes but also the names of properties. This is quite useful in several practical cases where Portal schemas may be composed of a) just property names (e.g., the Dublin Core Metadata Elements [50]) defined on the root class; b) large class hierarchies with only few properties defined between the top classes (e.g., when extending ontology concepts with thesauri terms [5]); or c) large property hierarchies resulting from the interconnection of several RDF schemas (e.g., when integrating different Metadata Schemas [23]). In such cases, the labels of nodes may not be available or users may not be aware of them. Still, RQL allows one to formulate queries as for example:

```
creates
```

By considering properties as binary relations, the above query will return the bag of ordered pairs of resources belonging to the extent of *creates* (*source* and *target* are simple position indices):

source	*target*
www.culture.net#rodin424	www.artchive.com/crucifixion.jpg
www.culture.net#picasso132	www.museum.es/guernica.jpg
www.culture.net#picasso132	www.museum.es/woman.qti

We can observe that, in the extent of properties we also consider their subproperties (e.g., *paints* and *sculpts*). We believe that using only few abstract labels (i.e., the top-level classes or properties in an RDF schema) to query complex descriptions is an original feature of RQL. As we will show in the sequel, properties are the main building blocks for formulating RQL path expressions. Common set operators applied to collections of the same type are also supported. For example, the query:

will return a bag with the URI www.artchive.com/crucifixion.jpg, since, according to our example, it is the only resource classified under both classes. Note that the typing system of *RQL* (see Appendix C for the typing system) permits the union of a bag of URIs with a bag of strings, but not between a class and a property extent (unary vs. binary relation).

Besides class or property extents, *RQL* also allows the manipulation of RDF container values as Bag and Sequence. More precisely, the Boolean operator `in` can be used for membership test in any kind of collection and the operator `element` is used to extract the unique member of a singleton. Additionally, to access a member of a Sequence we can use the operator "`[ ]`" with an appropriate position index (or index range). If the specified member elements do not exist the query will return an empty sequence.

Finally, *RQL* supports standard Boolean predicates as `=`, `<`, `>` and `like` (for string pattern matching). All operators can be applied on literal values or resource URIs. It should be stressed that the latter case also covers comparisons between class or property URIs. For example, the condition "Painter `<` Artist" will return `true` since the first operand is a subclass of the second one. Disambiguation is performed in each case by examining the operands (e.g., literal value vs. URI equality, lexicographical ordering vs. class ordering, etc.).

4.2 Personalizing Portal Access using RQL Filters

In order to personalize access to Community Web Portals, more complex *RQL* queries are needed. Portals personalization is actually supported by defining *information channels* to which community members may subscribe. Channels essentially preselect a collection of the Portal resources related to a theme, subject or topic (e.g., Museum Web sites) and they are specified using the recent RDF Site Summary (RSS) schema [7]. An RSS channel is specified by a static RDF/XML document containing the URIs of the resources along with some administrative metadata (e.g., titles, etc.). Not surprisingly, we can use *RQL* to define channels as views over the Portal Catalog and generate their contents on-demand.

In order to iterate over collections of RDF data (e.g., class or property extents, container values, etc.) and introduce variables, *RQL* provides a `select-from-where` filter. Given that the whole description base can be viewed as a collection of nodes/edges, *path expressions* can be used in *RQL* filters for the traversal of RDF graphs at arbitrary depths. The formal semantics of *RQL* filters and path expressions is given in Table 2. Consider, for instance, the following query:

Q1: *Find the resources having a title property.*

```
select  X, Y
from    {X}title{Y}
```

In the `from` clause we use a basic *data path expression* with the property name *title*. The node variables X and Y take their range restrictions from the *source* and *target* values of the *title* extent. As we can see in Figure 1, the *title* property has been defined with *domain* the class *ExtResource*

but, due to multiple classification, X may be valuated with resources also labeled with any other class name (e.g., *Artifact*, *Museum*, etc.). Yet, in our model X has the unique type τ_{UR}, Y the literal type *string*, and the result of **Q1** is of type $\{[\tau_{UR}, string]\}$. In **Q1** we essentially ignore the schema classes labeling the endpoint instances of the properties. Note that considering properties as entry-points to RDF graphs obviates the need to introduce more complex path expressions, featuring path variables, (in the style of POQL [2] or Lorel [3]) in order to access property values from a specific root entity (e.g., Artifacts{a}@P.title{t}).

The `select` clause defines, as usual, a projection over the variables of interest. Moreover, we can use "`select *`" to include in the result the values of all variables. This clause will construct an ordered tuple, whose arity depends on the number of variables. The result of the whole filter will be a bag. Thus, the type of the result of **Q1** will be $\{[\tau_{UR}, string]\}$. The closure of RQL is ensured by the basic queries supported for container values (see Table 1).

In order to define a channel with Museum resources available in our Cultural Portal we need to restrict the *source* values of the *title* extent to resources labeled with the class name *Museum* or any of its subclasses. To achieve this, we can formulate the following query:

Q2: *Find the Museum resources and their title.*

```
select  X, Y
from    Museum{X}.title{Y}
```

Here $Museum\{X\}$ is also a basic data path expression where X ranges over the resource URIs in the extent of class *Museum*. The "." used to concatenate the two path expressions is just a *syntactic shortcut* for an implicit join condition between the *source* values of the *title* extent and X. Hence, **Q2** is equivalent to the query $Museum\{X\}, \{Z\}title\{Y\}$ where $X = Z$. Recall that RDF classes do not define types on which attribute extractor operators like "." could be defined and therefore the expression $X.title$ is meaningless in out setting. Variables X, Z are of type τ_{UR}, Y is of type *string* and the result of **Q2** will contain all the resources accessible by our Museum channel along with their titles (e.g., the site `www.museum.es` with title "Reina Sofia Museum").

In addition, for each available resource (called item), channels usually provide a textual description of their information content. This description can also be generated automatically by appropriate RQL queries. For instance, we can use the names of artists whose artifacts are exhibited in the Museums as descriptions of our channel items. Hence, we need to formulate the following more complex query:

```
select  Y, Z, V, R
from    {X}creates.exhibited{Y}.title{Z}, {W}fname{V}, {Q}lname{R}
where   X = W and X = Q
```

In the `from` clause we use three data path expressions. Variable X (Y) ranges over the *source* (*target*) values of the *creates* (*exhibited*) property. The condition $X = W$ ($X = Q$) denotes an explicit join between the extents of the properties *fname* (*lname*) and *creates* on their *source*

Data Path:	**select** X **from** $c\{X\}$	$\{v \mid v \in [\![c]\!]\}$
	select X, Y **from** $\{X\}p\{Y\}$	$\{< v_1, v_2 > \mid < v_1, v_2 > \in [\![p]\!]\}$
	select $X, @P, Y$ **from** $\{X\}@P\{Y\}$	$\{< v_1, p, v_2 > \mid \exists p \in P, < v_1, v_2 > \in \hat{} [\![p]\!]\}$
Schema path:	**select** X **from** $Class\{X\}$	$\{c \mid c \in C\}$
	select P **from** $Property\{P\}$	$\{p \mid p \in P\}$
	select $\$C$ **from** $c\{:\$C\}$	$\{c' \mid c' \in C, c' \preceq c\}$
	select $\$X, \$Y$ **from** $\{:\$X\}p\{:\$Y\}$	$\{< c_1, c_2 > \mid c_1, c_2 \in C, c_1 \preceq domain(p), c_2 \preceq range(p)\}$
	select $\$X, @P, \$Y$ **from** $\{:\$X\}@P\{:\$Y\}$	$\{< c_1, p, c_2 > \mid p \in P, c_1, c_2 \in C, c_1 \preceq domain(p), c_2 \preceq range(p)\}$
Mixed path:	**select** $X, \$C$ **from** $c\{X:\$C\}$	$\{< v, c' > \mid c' \in C, c' \preceq c, v \in \hat{} [\![c']\!]\}$
	select $X, Y, \$Z, \$W$ **from** $\{X:\$Z\}p\{Y:\$W\}$	$\{< c_1, v_1, c_2, v_2 > \mid c_1, c_2 \in C, c_1 \preceq domain(p), v_1 \in \hat{} [\![c_1]\!], c_2 \preceq range(p), v_2 \in \hat{} [\![c_2]\!], < v_1, v_2 > \in [\![p]\!]\}$
	select $X, Y, @P, \$Z, \$W$ **from** $\{X:\$Z\}@P\{Y:\$W\}$	$\{< c_1, v_1, p, c_2, v_2 > \mid p \in P, c_1, c_2 \in C, c_1 \preceq domain(p), v_1 \in \hat{} [\![c_1]\!], c_2 \preceq range(p), v_2 \in \hat{} [\![c_2]\!], < v_1, v_2 > \in \hat{} [\![p]\!]\}$

Table 2: Formal Interpretation of the basic *RQL* path expressions

values. Since the *range* of the *exhibited* property is the class *Museum* we don't need to further restrict the labels for the Y values.

Two remarks are noteworthy. First, *RQL* data path expressions may be liberally composed in the `from` clause of a query using node and edge labels (see Table 3 for formal definitions) as for instance *Museum.tiltle* (a class and a property name) or *creates.exhibited* (two property names). The "." syntactic sugaring is used to introduce appropriate join conditions between the left and the right part of the expression (recall that RDF properties are directed). In the above query, it implies a join between the extents of *creates* and *exhibited* on their *target* and *source* values respectively. This way, *RQL* captures the existential semantics of navigation in semistructured RDF graphs[12]: there exist two *paints* properties for `www.culture.net#picasso132` while there is no *exhibited* property for `www.museum.es/woman.qti` (see Figure 1). Observe also that in the extent of *creates* we also consider its subproperties *paints* and *sculpts*.

Second, due to multiple classification of nodes (e.g., `www.museum.es` is both a *Museum* and *ExtResource*) we can query paths in a data graph that are not explicitly declared in the schema. For instance, *creates.exhibited.title* is not a valid schema path since the *domain* of the title property is the class *ExtResource* and not *Museum*. Still, we can query the coresponding data paths in the style of semistructured or XML query languages (e.g., LOREL [3], StruQL [25], XML-QL [24], XML-GL [14], Quilt [20]). However, as we will see in the next subsection, *RQL* is also able to filter data paths using in a more strict way schema information. Finally, the result of the previous query can be expressed in the RDF/XML syntax, in order to list the resources of our MuseumChannel, as follows:

```
1   <rdf:Bag ID="MuseumChannel">
```

[12] Nested queries featuring existential and universal quantification are also supported.

```
2     <rdf:li>
3        <rdf:Seq>
4           <rdf:li  rdf:resource = "www.museum.es"/>
5           <rdf:li> "Reina Sofia Museum"</rdf:li>
6           <rdf:li> Pablo </rdf:li>
7           <rdf:li> Picasso </rdf:li>
8        </rdf:Seq>
9     </rdf:li>
10    <rdf:li>
11       <rdf:Seq>
12          <rdf:li rdf:resource = "www.rodin.fr"/>
13          <rdf:li> "Rodin Museum"</rdf:li>
14          <rdf:li> August </rdf:li>
15          <rdf:li> Rodin </rdf:li>
16       </rdf:Seq>
17    </rdf:li>
18 </rdf:Bag>
```

For reasons of simplicity, in the remainder of the paper, we will present query results using an internal tabular representation (e.g., as ¬1NF relations), instead of RDF containers.

4.3 Querying Portals with Large Schemas

In the previous subsections, we have illustrated how RQL can be used to specify, in a declarative way, the access functionality actually supported by Portals like Netscape Open Directory. However, such simple browsing interfaces force the user to navigate through the whole hierarchy of topics (i.e., classes) in order to find resources classified under the leaf topics. It is evident that for large Portal schemas this is a cumbersome and time consuming task (e.g., the Art hierarchy of the Open Directory alone contains 20000 subtopics and currently 200000 indexed resources). Clearly, we also need declarative query support for navigating through the schema taxonomies of classes and properties. Consider, for instance, the following query:

Q3: *Find the resources of a type more specific than Painter and more general than Neo-Impressionist which have created something.*

`select` X, Y

`from` {X:$Z}creates{Y}

`where` $Z <= Painter `and` $Z >= Neo-Impressionist

In the from clause of **Q3** we can see a *mixed path expression* featuring both data (e.g., X) and schema variables on graph nodes (e.g., $\$Z$). More precisely, class variables prefixed by the symbol \$ are implicitly range restricted to `Class` (i.e., $Class\{Z\}$). Then, $\$Z$ is of type τ_{UC} and it will be valuated to the domain class of the property *creates* (i.e., `Artist`) and recursively to all of its subclasses (i.e., `Painter`, `Sculptor`, or `Neo-Impressionist`). The conditions in the `where` clause will in turn restrict $\$Z$ to the classes in the hierarchy having as superclass `Painter` and as subclass `Neo-Impressionist`[13]. Naturally, without any restriction to $\$Z$ the whole extent of *creates* will

[13] Note than in case of multiple inheritance, several paths in the class hierarchy will be traversed.

be returned and $\$Z$ will be valuated to the actual classes of its *source* values. Note that if the class in the `where` clause is not a valid subclass of the domain of *creates* then the query will return an empty bag without accessing the *creates* extent. To make this kind of path expressions more compact for class equality (e.g., $\$Z = Painter$), shortcuts as "`{X:Painter}creates{Y}`" are also supported. The formal semantics of RQL mixed path expressions are given in Table 2.

In other words, RQL extends the notion of generalized path expressions [15, 16, 3] to entire class (or property) inheritance paths. This is quite useful since resources can be multiply classified and several properties coming from different class hierarchies may be used for the same resource descriptions. Still RQL allows one to query properties emanating from resources only according to a specific class hierarchy, as we will see in the next example.

Q4: *Find the values of properties emanating from resources of type ExtResource.*

```
select   X,Y
from     {X:ExtResource}@P{Y}
```

X	Y
www.artchive.com/crucifixion.jpg	"image/jpg"
www.rodin.fr	"Rodin Museum"
www.museum.es	"Reina Sophia Museum"
www.rodin.fr	2000/06/09
www.museum.es	2000/02/01

The mixed path expression of **Q4**, features a schema variable on graph edges (e.g., $@P$). More precisely, property variables, prefixed by the symbol @, are implicity range restricted to `Property` (i.e., $Property\{P\}$). Then, $@P$ is of type τ_{UP} and it will be valuated to all properties having as domain `ExtResource` or one of its superclasses. Finally, X, Y will be range restricted for each successful binding of $@P$. The type of X is τ_{UR} while that of Y is the union of all the range types of `ExtResource` properties. According to the schema of Figure 1, $@P$ will valuated to *file_size*, *title*, *mime_type*, and *last_modified*, while Y will be of type $(integer + string + date)$. In case we want to filter Y values in the `where` clause, RQL supports appropriate coercions of union types in the style of POQL [2] or Lorel [3] (see Appendix C).

4.4 Querying Portal Schemas

In this subsection, we focus our attention on querying RDF schemas, regardless of any underlying instances. The main motivation for this is to use RQL as a high-level language to implement schema browsing. This is quite useful when Portal Catalogs use large schemas (e.g., the Open Directory Topic hierachy) to describe resources. In this context, Portal administrators may not be aware of all the classes and properties they can use to describe resources while RDF schemas carry information which is only implicitly stated (e.g., the polymorphism of the domain and range of properties). Consider for instance the following query:

Q5: *Find all the properties which specialize the property creates and may have as domain the class Painter along with their corresponding domain and range classes.*

```
select   @P, $Y
from     {:Painter}@P{:$Y}
where    @P <= creates
```

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

The *schema path expression* in the from clause of **Q5** introduces two variables: $@P$ (of type τ_{UP}) ranging over `Property`, and $\$Y$ (of type τ_{UC}) ranging over the range class (and its subclasses) of each $@P$ valuation ($\$Y <= range(@P)$). Furthermore, $@P$ should be a subproperty of *creates* for which the domain is *Painter* or one of its superclasses. This expression is just a shortcut for $\{:\$X\}@P\{:\$Y\}$ where $\$X = Painter$ and $\$X <= domain(@P)$. Given the schema of Figure 1, $@P$ will be valuated to the properties *creates* and *paints*. Due to class inheritance, *creates* may have as range any subclass of *Artifact*. The same is true for the range classes of *paints*. The formal semantics of *RQL* schema path expressions is given in Table 2.

In cases where an automatic expansion of class hierarchies is not desired, *RQL* allows one to obtain only the classes which are directly involved in the definition of properties. We can issue, for instance, the following query:

Q6: *For all the classes in the hierachy rooted at Artist find the directly defined properties and their domain and range classes.*

```
select   domain(@P), @P, range(@P)
from     Property{@P}
where    domain(@P) <= Artist
```

$domain(@P)$	$@P$	range(@P)
Artist	creates	Artifact
Artist	fname	string
Artist	lname	string
Painter	paints	Painting
Sculptor	sculpts	Sculpture

Compared to **Q5** the result of this query will contain only the classes for which a property is defined, along with its name and range. Note also that the functional nature of *RQL* allows the use of functions anywhere in a filter as long as typing rules are respected: `domain(@P)` is of type τ_{UC}, as is the class name *Artist* and `range(@P)` is of type τ_{UCT} (i.e., the URIs of classes extended to include type names).

We conclude this subsection, with a query example illustrating how *RQL* schema paths can be composed to perform more complex schema navigations (see Table 3 for formal definitions). Note that this kind of queries cannot be expressed in existing languages with schema querying capabilities (e.g., SchemaSQL [32], XSQL [31], Noodle [40]).

Q7: *What properties can be reached (at one step) from the range classes of creates.*

`select` X, Y `from` $c\{X\}.p\{Y\}$	$\{< v_1, v_2 > \mid v_1 \in [\![c]\!], < v_1, v_2 > \in [\![p]\!]$
`select` X, Y, Z `from` $\{X\}p\{Y\}.q\{Z\}$	$\{< v_1, v_2, v_3 > \mid < v_1, v_2 > \in [\![p]\!], < v_2, v_3 > \in [\![q]\!]\}$
`select` $\$X, \$Y, \$Z$ `from` $\{: \$X\}p\{: \$Y\}.q\{: \$Z\}$	$\{< c_1, c_2, c_4 > \mid c_1, c_2, c_3, c_4 \in C, c_1 \preceq domain(p), c_2 \preceq range(p), c_3 \preceq domain(q), c_4 \preceq range(q), c_2 = c_3\}$
`select` $X, \$C_1, Y, \$C_2, Z, \$C_4$ `from` $\{X : \$C_1\}p\{Y : \$C_2\}.q\{Z : \$C_4\}$	$\{< v_1, c_1, v_2, c_2, v_4, c_4 > \mid c_1, c_2, c_3, c_4 \in C, c_1 \preceq domain(p), v_1 \in \hat{}[\![c_1]\!], c_2 \preceq range(p), v_2 \in \hat{}[\![c_2]\!], c_3 \preceq domain(q), v_3 \in \hat{}[\![c_3]\!], c_4 \preceq range(q), v_4 \in \hat{}[\![c_4]\!], < v_1, v_2 > \in [\![p]\!], < v_3, v_4 > \in [\![q]\!], c_2 = c_3, v_2 = v_3\}$
`select` $X, \$C_1, Y, \$C_2, Z$ `from` $\{X : \$C_1\}p\{Y : \$C_2\}.q\{Z\}$	$\{< v_1, c_1, v_2, c_2, v_4 > \mid c_1, c_2 \in C, c_1 \preceq domain(p), v_1 \in \hat{}[\![c_1]\!], c_2 \preceq range(p), v_2 \in \hat{}[\![c_2]\!], < v_1, v_2 > \in [\![p]\!], < v_3, v_4 > \in [\![q]\!], v_2 = v_3\}$
`select` $X, \$C_1, Y, \$C_2, \$C_4$ `from` $\{X : \$C_1\}p\{Y : \$C_2\}.q\{: \$C_4\}$	$\{< v_1, c_1, v_2, c_2, c_4 > \mid c_1, c_2, c_3, c_4 \in C, c_1 \preceq domain(p), v_1 \in \hat{}[\![c_1]\!], c_2 \preceq range(p), v_2 \hat{} \in [\![c_2]\!], < v_1, v_2 > \in [\![p]\!], c_3 \preceq domain(q), c_4 \preceq range(q), c_2 = c_3\}$
`select` $X, Y, \$C_4$ `from` $\{X\}p\{Y\}.q\{: \$C_4\}$	$\{< v_1, v_2, c_4 > \mid c_3, c_4 \in C, < v_1, v_2 > \in [\![p]\!], c_3 \preceq domain(q), c_4 \preceq range(q), v_2 \in \hat{}[\![c_3]\!]\}$

Table 3: Formal Interpretation of RQL Path compositions

`select` $Y, @P, range(@P)
`from` creates{:$Y}.@P

$Y	@P	range(@P)
Artifact	exhibited	Museum
Painting	exhibited	Museum
Sculpture	exhibited	Museum
Painting	technique	string
Sculpture	material	string

In **Q7** the "." notation implies a join condition between the range classes of the property *creates* and the domain of $@P$ valuations: $\$Y <= domain(@P)$. As we can see from the query result, this join condition will enable us to follow properties which can be applied (i.e., either directly defined of inherited) to any subclass of the *creates* range.

4.5 Putting it all Together

In the previous subsections we have presented the main RQL path expressions allowing us to browse and filter description bases with or whithout schema knowledge, or, alternatively to query exclusively the description schemas. As we will see in the sequel, RQL filters also admit arbitrary mixtures of different kinds of path expressions (see Table 3 for formal definitions). This functionality is required especially when different servers of Portal Catalogs (within or across communities) need to exchange resource metadata. The following two scenarios depict this functionality.

In the first, we assume two servers for Portal Catalogs having different schemas (e.g., identified by two namespaces $ns1$ and $ns2$ shown in Figure 1) while sharing the same description base (e.g., in the case of sub-communities). Then we can combine schema information from the first server with data information from the second one, as illustrated in the next example:

Q8: *Find the resources modified after 2000/01/01 which can be reached by a property applied to the class Painting and its subclasses.*

```
select   R, Y, Z
from     (select @P
          from  {:$X}@P
          where $X <= Painting
         ){R}.{Y}last_modified{Z}
where    Z > 2000/01/01
```

R	Y	Z
exhibited	www.museum.es	2000/06/09
exhibited	www.rodin.fr	2000/02/01

In **Q8** the nested query will return all the property names which are used by the first server and satisfy the filtering conditions (e.g., *exhibited*, *technique*). Then, the result of this nested query will serve to query the description base using the properties known only by the second server (e.g., *last_modified*). Here, the variable R (of type τ_{UP}) will iterate over the result of the nested query and the shortcut "." implies the join condition: $target(R) = Y$ for each valuation of R. In other terms, we will obtain those resources modified after 2000/01/01 for which there exists an incoming edge with a label (property) that is returned by the nested query.

In the second scenario, the two Portal servers have both different description bases and schemas. Then one of them can send to the other the following query:

Q9: *Tell me everything you know about the resources of the site "www.museum.es".*

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

This query will iterate over all property names ($@P$). Then for each property it will iterate over its possible domain ($\$\$Z$) and range ($\$\$W$) classes or types, and, finally over the corresponding extents (X, Y). We use the prefix $$ to denote variables ranging over both class and type names (i.e., of type τ_{UCT}). According to the example of Figure 1 the type of Y is the union ($\tau_{UC} + string + integer + date$) and the predicate `like` will be applied only on class names and strings. The final result of **Q9** is given below:

X	$\$\$Z$	$@P$	Y	$\$\$W$
www.culture.net#picasso132	Painter	paints	www.museum.es/guernica.jpg	Painting
www.culture.net#picasso132	Painter	paints	www.museum.es/woman.qti	Painting
www.museum.es/guernica.jpg	Painting	exhibited	www.museum.es	Museum
www.museum.es/guernica.jpg	Painting	technique	"oil on canvas"	string
www.museum.es/woman.qti	Painting	technique	"oil on canvas"	string
www.museum.es	ExtResource	title	"Reina Sophia Museum"	string
www.museum.es	ExtResource	last_modified	2000/06/09	date

We conclude this section with a brief comparison of *RQL* to other related languages, and in particular logic-based frameworks for RDF querying. SiLRI [21] proposes some reasoning mechanisms to query RDF descriptions and schemas using F-logic. Although powerful, this approach does not capture the peculiarities of RDF: refinement of properties is not allowed (since slots are

locally defined within the classes), container values for modeling n-ary relations are not supported (since it relies in a pure object model), while resource descriptions having heterogeneous types is not possible (due to strict typing). In the opposite direction, Metalog [38] uses Datalog to model mainly RDF properties as binary predicates, while it suggests an extension of the RDFS specification with variables and logical connectors (and, or, not, implies). However, storing and querying RDF descriptions with Metalog almost totally disregards RDF schemas. None of the existing proposals combines all the functionality we provide with *RQL*.

5 The RQL Interpreter and RDF Storage System

As we have seen in the previous section, the algebraic interpretation of *RQL*, relies on a relational representation of RDF description and schema graphs. Hence, we have implemented RDF storage and querying on top of an object-relational DBMS (ORDBMS), namely PostgreSQL[14]. The architecture of our RDF-enabled DBMS is illustrated in Figure 2. It comprises three main components: the RDF validator and loader (**VRP**), the RDF description database (**DBMS**) and the query language interpreter (**RQL**).

5.1 RDF Loader and Object-Relational Representation

We have implemented our *loader* as an extension of the *Validating RDF Parser* (VRP[15]) for analyzing, validating and processing RDF schemas and descriptions. Unlike other RDF parsers (e.g., SiRPAC[16]), VRP 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 *VRP validation module* relies on an internal object representation, separating RDF schemas from their instances. This representation simplifies RDF metadata manipulation while it enables an *incremental loading* of RDF descriptions and schemas which is crucial for large volumes of RDF data (e.g., Netscape Open Directory exports 100M of class hierarchies and 700M of resource descriptions). The various loading methods have been implemented as member functions of the related VRP internal classes and communication with PostgreSQL relies on the JDBC protocol.

The core RDF/S model is represented in PostgreSQL by four tables, namely, `Class`, `Property`, `SubClass` and `SubProperty` which capture the class and property-type hierarchies defined in an RDF schema. Then, for every class or property used in a Portal Catalog we create a new table to store its instances (recall that all names are unique). In other words, our RDF-enabled DBMS relies on a schema specific representation of resource descriptions similar to the *attribute-based* approach proposed for storing XML data [27, 49]. This approach ensures better query performances and smaller database volumes for storing (and indexing) RDF data than others using a monolithic

[14] www.postgresql.org
[15] www.ics.forth.gr/proj/isst/RDF
[16] www.w3.org/RDF/Implementations/SiRPAC

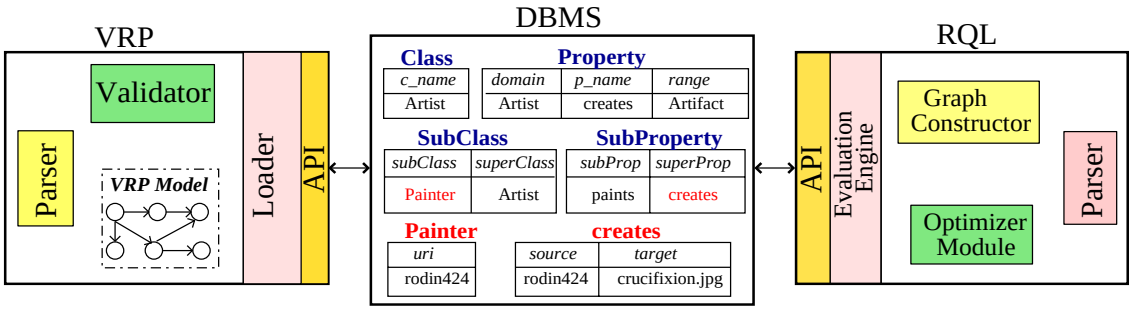

Figure 2: The RQL Interpreter and Storage System

table [39, 35] to represent RDF descriptions and schemas under the form of triples.

Three remarks are noteworthy. First, unlike XML, RDF graphs contains labels for both graph nodes and edges. Therefore, we need to generate tables for both properties and class instances. Second, RDF labels may be organized in taxonomies through multiple specialization (in opposite to element types defined in XML DTDs or Schemas). This information is captured by the `SubClass` and `SubProperty` tables, while the corresponding instance tables are also connected through the *subtable* relationship, supported by PostgreSQL. Third, there is no real need for expensive (due to data fragmentation in several tables) *RQL* queries, reconstructing the initially loaded resource descriptions - as in XML query languages - since there are various ways to serialize RDF graph descriptions in a forest of XML trees (see Appendix A). As a matter of fact, several variations (similar to [47]) of the above relational representation are currently studied in order to minimize the number of joins required to evaluate RQL path expressions.

5.2 RQL Query Processing

The *RQL interpreter* consists of (a) the parser, analyzing the syntax of queries; (b) the graph constructor, capturing the semantics of queries in terms of typing (see Appendix C) and inter-dependencies of involved expressions; and (c) the evaluation engine, accessing RDF descriptions from the underlying database. Since our implementation relies on a full-fledged ORDBMS like PostgreSQL, the goal of the *RQL optimizer* is to push as much as possible query evaluation to the underlying SQL3 engine (communication is based on libpq++, a JDBC-level, C++ PostgreSQL API). Then pushing selections or reordering joins to evaluate *RQL* path expressions is left to PostgreSQL while the evaluation of *RQL* functions for traversing class and property hierarchies relies on the existence of appropriate indices (see the last paragraph). The main difficulty in translating an entire *RQL* algebraic expression (expressed in an object algebra a la [19]) to a single SQL3 query is due to the fact that most *RQL* path expressions interleave schema with data querying [16]. Consider for instance the following query:

```
select  y
from    {x}creates{y:Painting}.technique{z}
where   z="oil on canvas"
```

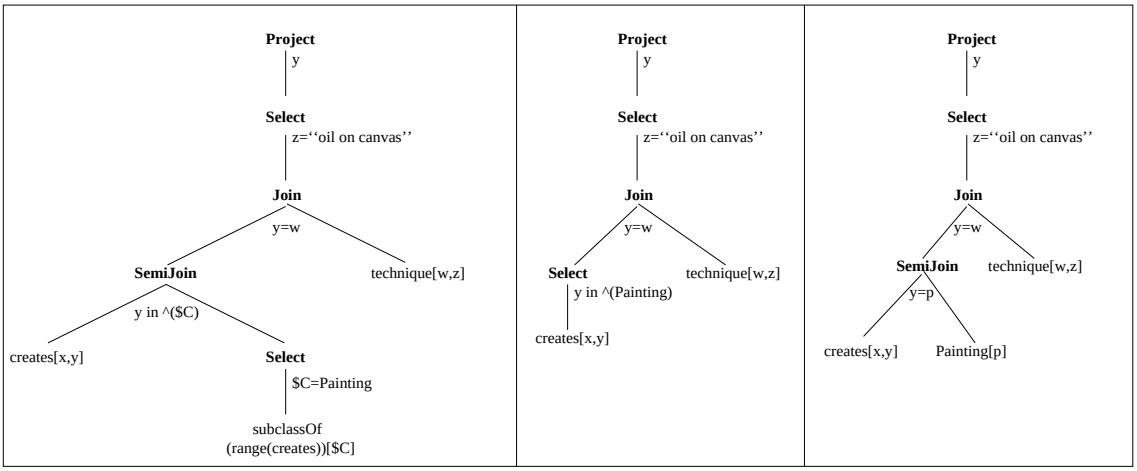

Figure 3: Example of an RQL query optimization

During the query graph construction, the various shortcuts (e.g., ".") are expanded and variable dependencies are determined. The algebraic translation of the above query is illustrated in the left part of Figure 3. Data variables x, y (w, z) iterate over the extent of *creates* (*technique*) while the class variable $\$C$ iterates over all the subclasses of the range of *creates*. The mixed path expression of our example is translated to a semi-join between the extent of *creates* and its permitable range classes (y in $\hat{}\$C$). Since in this query we are interested only in instances of the class *Painting* we can omit the second branch of the semi-join and rewrite the algebraic expression as illustrated in the middle part of Figure 3 (semi-joins can be always transformed to selections, products/joins and vice-versa). Then the selection implies an existential condition over the extent of *Painting*. This operation is equivalent to a semi-join over *creates* and *Painting* as illustrated in the right part of Figure 3. The final expression is translated into an SQL3 query as follows (the * indicates an extended interpretation of tables, according to the subtable hierarchy):

```
select X.target
from   creates* X, technique* Y, Painting P
where  X.target = Y.source and X.target = P.uri and Y.target='oil on canvas'
```

We conclude this section with one remark concerning the encoding of class and property names. Recall that schema or mixed *RQL* path expressions need to recursively traverse a given class (or property) hierarchy. We can transform such traversal queries into interval queries on a linear domain, that can be answered efficiently by standard DBMS index structures (e.g., B-trees). For this, we need to replace class (or property) names by *ids* using an appropriate encoding system (e.g., Dewey, postfix, prefix, etc.) for which a convenient total order exists between the elements in the hierarchy. We are currently working on the choice of a such linear representation of node or edge labels allowing us to optimize queries that involve different kinds of traversals in a hierarchy (e.g., an entire subtree, a path from the root, etc.).

6 Summary and Future Work

In this paper, we presented a declarative language for querying Portal Catalogs, through a series of examples of increasing expressiveness requirements and complexity. The novelty of *RQL* lies in its ability to smoothly combine schema and data querying, while transparently exploiting taxonomies of classes and properties as well as multiple classification of resources. We believe that we have illustrated the power of RQL generalized path expressions (featuring regular expressions on labels of both nodes and edges), subsuming the filtering capabilities (i.e., without any restructuring operators) of semi-structured and XML query languages [3, 25, 24, 14, 20], as well as schema query languages [32, 31, 40]. Due to its functional nature, RQL can also express the majority[17] of queries expressible in the series of languages presented for querying network directories [30]. A detailed analysis of the evaluation complexity of the language is an ongoing effort. Furthermore, exhaustive performance tests of *RQL* for various use cases (as the Open Directory and our home made Cultural Portal) are currently under elaboration taking into account different variations of our relational representation, as well as, different encoding schemas for class and property hierarchies. Finally, we are also planning to study the problem of *updates* in RDF description bases as well as the *restructuring* capabilities (e.g., grouping) of *RQL* required by various Community Web Portal applications.

Acknowledgments

We would like to thank Jeen Broekstra and Arjohn Kampman for their valuable comments on the syntax of *RQL*, Alain Michard for many fruitful discussions on Community Web Portals, Jérôme Siméon for understanding the subtleties between RDF and XML schema and Manolis Koubarakis for carefully reading and commenting previous versions of this paper.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[2] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Siméon. Querying Documents in Object Databases. *International Journal on Digital Libraries*, 1(1):5–18, April 1997.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[4] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[5] B. Amann and I. Fundulaki. Integrating Ontologies and Thesauri to Build RDF Schemas. In *ECDL-99: Research and Advanced Technologies for Digital Libraries*, Lecture Notes in Computer Science, pages 234–253, Paris, France, September 1999. Springer-Verlag.

[17] With the exception of count, we have not yet considered the addition of canonical aggregation operators such as max, min etc. in RQL.

[6] S. Amer-Yahia, H. Jagadish, L. Lakshmanan, and D. Srivastava. On bounding-schemas for ldap directories. In *Proc. of the International Conf. on Extending Database Technology*, pages 287–301, Konstanz, Germany, March 2000.

[7] G. Beged-Dov, D. Brickley, R. Dornfest, I. Davis, L. Dodds, J. Eisenzopf, D. Galbraith, R. Guha, E. Miller, and E. van der Vlist. RSS 1.0 Specification Protocol. Draft, August 2000.

[8] T. Bray, J. Paoli, and C.M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C Recommendation, February 1998. Available at http://www.w3.org/TR/REC-xml/.

[9] D. Brickley and R.V. Guha. Resource Description Framework (RDF) Schema Specification 1.0, W3C Candidate Recommendation. Technical Report CR-rdf-schema-20000327, W3C, Available at http://www.w3.org/TR/rdf-schema, March 27, 2000.

[10] A. Brooking. *Corporate Memory: Strategies For Knowledge Management.* International Thomson Business Press, 1998.

[11] P. Buneman, S.B. Davidson, and D. Suciu. Programming Constructs for Unstructured Data. In *Proc. of International Workshop on Database Programming Languages*, Gubbio, Italy, 1995.

[12] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, 1988.

[13] R.G.G. Cattell and D. Barry. *The Object Database Standard ODMG 2.0.* Morgan Kaufmann, 1997.

[14] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: a Graphical Language for Querying and Restructuring XML Documents. In *Proc. of International World Wide Web Conf.*, Toronto, Canada, 1999.

[15] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 313–324, Minneapolis, Minnesota, May 1994.

[16] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating Queries with Generalized Path Expressions. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 413–422, Montreal, Canada, June 1996.

[17] V. Christophides, S. Cluet, and J. Siméon. On Wrapping Query Languages and Efficient XML Integration. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Dallas, TX., May 2000.

[18] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your Mediators Need Data Conversion! In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 177–188, Seattle, WA., June 1998.

[19] S. Cluet and G. Moerkotte. Nested Queries in Object Bases. In *DBPL'93*, pages 226–242, 1993.

[20] D. Florescu D. Chamberlin, J. Robie. Quilt: An xml query language for heterogeneous data sources. In *WebDB'2000*, pages 53–62, Dallas, US., May 2000.

[21] S. Decker, D. Brickley, J. Saarela, and J. Angele. A query and inference service for rdf. In *W3C Query Languages Workshop*, Cambridge, Mass., 1998.

[22] L. Delcambre and D. Maier. Models for superimposed information. In *ER '99 Workshop on the World Wide Web and Conceptual Modeling*, volume 1727 of *Lecture Notes in Computer Science*, pages 264–280, Paris, France, November 1999. Springer.

[23] L. Dempsey and R. Heery. DESIRE: Development of a European Service for Information on Research and Education, 1997. http://www.ukoln.ac.uk/metadata/desire/-overview/rev_ti.htm.

[24] A. Deutsch, M.F. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *Proc. of International World Wide Web Conf.*, Toronto, 1999.

[25] M.F. Fernandez, D. Florescu, J. Kang, A.Y. Levy, and D. Suciu. System Demonstration - Strudel: A Web-site Management System. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Tucson, AZ., May 1997. Exhibition Program.

[26] C. Finkelstein and P. Aiken. *Building Corporate Portals using XML.* McGraw-Hill, 1999.

[27] D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing xml data in a relational database. Technical Report 3680, INRIA Rocquencourt, France, May 1999. Available at http://www-caravel.inria.fr/dataFiles/GFSS00.ps.

[28] D.J. Foskett. Theory of clumps. In K. Sparck Jones and P. Willett, editors, *Readings in Information Retrieval*, pages 111–134. Morgan Kaufmann, 1997.

[29] ISO. Information Processing-Text and Office Systems- Standard Generalized Markup Language (SGML). ISO 8879, 1986.

[30] H. Jagadish, L. Lakshmanan, T. Milo, D. Srivastava, and D. Vista. Querying network directories. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 133–144, Philadelphia, USA, 1999. ACM Press.

[31] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 393–402, 1992.

[32] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. SchemaSQL - a language for interoperability in relational multi-database systems. In *Proc. of International Conf. on Very Large Databases (VLDB)*, pages 239–250, Bombay, India, September 1996.

[33] O. Lassila and R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. Technical report, World Wide Web Consortium, February 1999. Available at http://www.w3.org/TR/REC-rdf-syntax.

[34] J. Liebowitz. *Building Organizational Intelligence: A Knowledge Management Primer*. CRC Press, Boca Raton, FL., 1999.

[35] J. Liljegren. Description of an rdf database implementation. Available at http://WWW-DB.stanford.edu/~melnik/rdf/db-jonas.html.

[36] D. Maier and L. Delcambre. Superimposed information for the internet. In *ACM SIGMOD Workshop on The Web and Databases Philadelphia, Pennsylvania, June 3-4*, pages 1–9, 1999.

[37] M. Maloney and A. Malhotra. XML schema part 2: Datatypes. W3C Candidate Recommendation, October 2000. Available at http://www.w3.org/TR/xmlschema-2/.

[38] M. Marchiori and J. Saarela. Query + metadata + logic = metalog. In *W3C Query Languages Workshop*, Cambridge, Mass., 1998.

[39] S. Melnik. Storing rdf in a relational database. Available at http://WWW-DB.stanford.edu/-~melnik/rdf/db.html.

[40] I.S. Mumick and K.A. Ross. Noodle: A Language for Declarative Querying in an Object-Oriented Database. In *Proc. of International Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 360–378, Phoenix, Arizona, December 1993.

[41] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing Knowledge about Information Systems. *ACM TOIS*, 8(4):325–362, 1990.

[42] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. MedMaker: A Mediation System Based on Declarative Specifications. In *Proc. of IEEE International Conf. on Data Engineering (ICDE)*, pages 132–141, New Orleans, LA., February 1996.

[43] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proc. of IEEE International Conf. on Data Engineering (ICDE)*, pages 251–260, Taipei, Taiwan, March 1995.

[44] D. Plexousakis. Semantical and Ontological Considerations in Telos: a Language for Knowledge Representation. *Computational Intelligence*, 9(1):41–72, 1993.

[45] A. Radding. *Knowledge Management: Succeeding in the Information-based Global Economy*. Computer Technology Research Corp., 1998.

[46] Some proposed RDF APIs.
GINF: http://www-db.stanford.edu/~melnik/rdf/api.html,
RADIX: http://www.mailbase.ac.uk/lists/rdf-dev/1999-06/0002.html,
Netscape Communicator: http://lxr.mozilla.org/seamonkey/source/rdf/base/idl/,
RDF for Java: http://www.alphaworks.ibm.com/formula/rdfxml/.

[47] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. DeWitt, and J.F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proc. of International Conf. on Very Large Databases (VLDB)*, pages 302–314, Edinburgh, Scotland, UK, September 1999. Morgan Kaufmann.

[48] H.S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML schema part 1: Structures. W3C Candidate Recommendation, October 2000. Available at http://www.w3.org/TR/-xmlschema-1/.

[49] F. Tian, D. DeWitt, J. Chen, and C. Zhang. The Design and Performance Evaluation of Alternative XML Storage Strategies. Technical report, CS Dept., Universiy of Wisconsin, 2000. Available at http://www.cs.wisc.edu/niagara/papers/vldb00XML.pdf.

[50] S. Weibel, J. Miller, and R. Daniel. Dublin Core. In *OCLC/NCSA metadata workshop report*, 1995.

A An Example of RDF Descriptions and Schemas

```
<?xml version="1.0"?>

<rdf:RDF xml:lang="en"
       xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
       xmlns:rdfs="http://www.w3.org/TR/2000/CR-rdf-schema-20000327#"
       xmlns="">
<rdfs:Class rdf:ID="Artist"/>
<rdfs:Class rdf:ID="Artifact"/>
<rdfs:Class rdf:ID="Museum"/>
<rdfs:Class rdf:ID="Sculptor">
      <rdfs:subClassOf rdf:resource="#Artist"/>
</rdfs:Class>
<rdfs:Class rdf:ID="Painter">
      <rdfs:subClassOf rdf:resource="#Artist"/>
</rdfs:Class>
<rdfs:Class rdf:ID="Sculpture">
      <rdfs:subClassOf rdf:resource="#Artifact"/>
</rdfs:Class>
<rdfs:Class rdf:ID="Painting">
      <rdfs:subClassOf rdf:resource="#Artifact"/>
</rdfs:Class>
<rdfs:Class rdf:ID="ExtResource"/>

<rdf:Property rdf:ID="creates">
      <rdfs:domain rdf:resource="#Artist"/>
      <rdfs:range rdf:resource="#Artifact"/>
</rdf:Property>
<rdf:Property rdf:ID="paints">
      <rdfs:domain rdf:resource="#Painter"/>
      <rdfs:range rdf:resource="#Painting"/>
      <rdfs:subPropertyOf rdf:resource="#creates"/>
</rdf:Property>
<rdf:Property rdf:ID="sculpts">
      <rdfs:domain rdf:resource="#Sculptor"/>
      <rdfs:range rdf:resource="#Sculpture"/>
      <rdfs:subPropertyOf rdf:resource="#creates"/>
</rdf:Property>
<rdf:Property rdf:ID="technique">
      <rdfs:domain rdf:resource="#Painting"/>
      <rdfs:range rdf:resource="http://www.w3.org/rdf-datatypes.xsd#String"/>
</rdf:Property>
<rdf:Property rdf:ID="material">
      <rdfs:domain rdf:resource="#Sculpture"/>
      <rdfs:range rdf:resource="http://www.w3.org/rdf-datatypes.xsd#String"/>
</rdf:Property>
<rdf:Property rdf:ID="fname">
      <rdfs:domain rdf:resource="#Artist"/>
      <rdfs:range rdf:resource="http://www.w3.org/rdf-datatypes.xsd#String"/>
</rdf:Property>
<rdf:Property rdf:ID="lname">
      <rdfs:domain rdf:resource="#Artist"/>
      <rdfs:range rdf:resource="http://www.w3.org/rdf-datatypes.xsd#String"/>
</rdf:Property>
<rdf:Property rdf:ID="exhibited">
      <rdfs:domain rdf:resource="#Artifact"/>
      <rdfs:range rdf:resource="#Museum"/>
</rdf:Property>
<rdf:Property rdf:ID="title">
      <rdfs:domain rdf:resource="#ExtResource"/>
```

```
      <rdfs:range rdf:resource="http://www.w3.org/rdf-datatypes.xsd#String"/>
</rdf:Property>
<rdf:Property rdf:ID="mime-type">
      <rdfs:domain rdf:resource="#ExtResource"/>
      <rdfs:range rdf:resource="http://www.w3.org/rdf-datatypes.xsd#String"/>
</rdf:Property>
<rdf:Property rdf:ID="file_size">
      <rdfs:domain rdf:resource="#ExtResource"/>
      <rdfs:range rdf:resource="http://www.w3.org/rdf-datatypes.xsd#Integer"/>
</rdf:Property>
<rdf:Property rdf:ID="last_modified">
      <rdfs:domain rdf:resource="#ExtResource"/>
      <rdfs:range rdf:resource="http://www.w3.org/rdf-datatypes.xsd#Date"/>
</rdf:Property>

<Painter rdf:ID="picasso132">
     <paints>
        <Painting rdf:about="http://www.museum.es/guernica.jpg">
          <technique>oil on canvas</technique>
            <exhibited>
              <Museum rdf:about="http://www.museum.es"/>
            </exhibited>
        </Painting>
     </paints>
     <paints>
        <Painting rdf:about="http://www.museum.es/woman.qti">
          <technique>oil on canvas</technique>
        </Painting>
     </paints>
     <fname>Pablo</fname>
     <lname>Picasso</lname>
</Painter>

<Sculptor rdf:ID="rodin424" fname="August" lname="Rodin">
     <creates>
       <Sculpture rdf:about="http://www.artchive.com/crucifixion.jpg">
       </Sculpture>
     </creates>
</Sculptor>

<Sculpture rdf:about="http://www.artchive.com/crucifixion.jpg">
     <exhibited> <Museum rdf:about="http://www.rodin.fr"/> </exhibited>
</Sculpture>

<ExtResource rdf:about="http://www.museum.es/guernica.jpg"/>
<ExtResource rdf:about="http://www.museum.es/woman.qti"/>
<ExtResource rdf:about="http://www.artchive.com/crucifixion.jpg">
     <mime-type>image/jpeg</mime-type>
</ExtResource>
<ExtResource rdf:about="http://www.museum.es">
     <title>Reina Sofia Museum</title>
     <last_modified>2000/06/09</last_modified>
</ExtResource>
<ExtResource rdf:about="http://www.rodin.fr">
     <title>Rodin Museum</title>
     <last_modified>2000/02/01</last_modified>
</ExtResource>

</rdf:RDF>
```

B The BNF Grammar of RQL

query ::= "(" *query* ")"
| "`subClassOf`" ["`^`"] "(" *query* ")"
| "`subPropertyOf`" ["`^`"] "(" *query* ")"
| "`domain`" "(" *query* ")"
| "`range`" "(" *query* ")"
| *query set_op query*
| *query bool_op query*
| "`not`" *query*
| *query comp_op query*
| *query* "`in`" *query*
| "`count`" "(" *query* ")"
| "`element`" "(" *query* ")"
| *query* "`[`" *query* ["`:`" *query*] "`]`"
| "`^`" *query*
| *constant*
| *var*
| identifier
| "`Class`"
| "`Property`"
| "`bag`(" *query*, [*query*]")"
| "`seq`(" *query*, [*query*]")"
| *sfw_query*
| "`exists`" *var query* "`:`" *query*
| "`forall`" *var query* "`:`" *query*

sfw_query ::= "`select`" *projslist* "`from`" *rangeslist* ["`where`" *query*]

comp_op ::= "`<`" | "`<=`" | "`>`" | "`>=`" | "`=`" | "`!=`" | "`like`"

set_op ::= "`union`" | "`intersect`" | "`minus`"

bool_op ::= "`and`" | "`or`"

constant ::= integer_literal
| real_literal
| quoted_string_literal
| quoted_char_literal
| date
| "`true`"
| "`false`"
| "`&`" identifier

var ::= *data_var* | *class_var* | *type_var* | *property_var*

data_var ::= identifier

class_var ::= "`$`" identifier

type_var ::= "`$`" "`$`" identifier

property_var ::= "`@`" identifier

projslist ::= "`*`" | *query* { "`,`" *query* }

rangeslist ::= *pathexpr* { "`,`" *pathexpr* }

pathexpr ::= *pathelem* { "`.`" *pathelem* }

pathelem ::= ["{" *from_to* "}"] *query* ["{" *from_to* "}"]

from_to ::= [*data_var*] ["`:`" (*class_var* | *type_var* | identifier)]

C RQL Typing System

C.1 Basic Queries

ith:
$$\frac{e : [\tau], i : integer, i \ \in \ [1..n]}{(e[i]) : \tau}$$

in:
$$\frac{e : \tau, e' : (\{\tau\} \mid [\tau])}{(e \ in \ e') : boolean}$$

comp:
$$\frac{e : \tau, e' : \tau', \tau = \tau', \theta \in (=, \neq, <, >, \leq, \geq like)}{(e \ \theta \ e') : boolean}$$

ucomp:
$$\frac{e : (\tau_1 + \tau_2 + \ldots + \tau_n), e' : \tau', \exists i \in [1..n] : \tau_i = \tau', \theta \in (=, \neq, <, >, \leq, \geq, like)}{(e \ \theta \ e') : boolean, coerce(e) : \Sigma_i(\tau_i)}$$

ucompu:
$$\frac{e : (\tau_1 + \tau_2 + \ldots + \tau_n), e' : (\tau'_1 + \tau'_2 + \ldots + \tau'_n), \exists i, j \in [1..n] : \tau_i = \tau'_j, \theta \in (=, \neq, <, >, \leq, \geq like)}{(e \ \theta \ e') : boolean, coerce(e) : \Sigma_i(\tau_i), coerce(e') : \Sigma_j(\tau'_j)}$$

set operations:
$$\frac{col_1 : (\{\tau\} \mid [\tau]), col_2 : (\{\tau'\} \mid [\tau']), \tau = \tau', \theta \in (intersect \mid union \mid minus)}{(col_1 \ \theta \ col_2) : \{\tau\}}$$

C.2 Filters and Path Expressions

data paths:
$$\frac{e : \{\tau\}}{(\text{select } x \text{ from } e\{x\}) : \{\tau\}, x : \tau}$$

$$\frac{e : \{[\tau_1, \tau_2]\}}{(\text{select } x, y \text{ from } \{x\}e\{y\}) : \{[\tau_1, \tau_2]\}, x : \tau_1, y : \tau_2}$$

$$\frac{\forall @e \in P, @e : \{[\tau_i, \tau'_i]\}}{(\text{select } x, y \text{ from } \{x\}@e\{y\}) : \{[\Sigma_i(\tau_i), \Sigma_i(\tau'_i)]\}, x : \Sigma_i(\tau_i), y : \Sigma_i(\tau'_i)}$$

---

schema paths:
$$\frac{e : \{\tau\}, \tau = (\tau_{UC} | \tau_{UP}), r = (\$c \mid @p)}{(\text{select } r \text{ from } e\{r\}) : \{\tau\}, r : \tau}$$

$$\frac{e : \tau_{UP}, domain(e) : \tau_{UC}, range(e) : \tau_{UCL}}{(\text{select } \$c_1, \$c_2 \text{ from } \{\$c_1\}e\{\$c_2\}) : \{[\tau_{UC}, \tau_{UCL}]\}, \$c_1 : \tau_{UC}, \$c_2 : \tau_{UCL}}$$

$$\frac{\forall @e \in P, domain(@e) : \tau_{UC}, range(@e) : \tau_{UCL}}{(\text{select } \$c_2, \$c_2 \text{ from } \{\$c_1\}@e\{\$c_2\}) : \{[\tau_{UC}, \tau_{UCL}]\}, \$c_1 : \tau_{UC}, \$c_2 : \tau_{UCL}}$$

---

mixed paths:
$$\frac{e : \tau_{UP}, domain(e) : \tau_{UC}, range(e) : \tau_{UCL}, eval(e) : \{[\tau_1, \tau_2]\}}{\begin{array}{c}(\text{select } \$c_1, x, \$c_2, y \text{ from } \{x : \$c_1\}e\{y : \$c_2\}) : \\ \{[\tau_{UC}, \tau_1, \tau_{UCL}, \tau_2]\}, \$c_1 : \tau_{UC}, x : \tau_1, \$c_2 : \tau_{UCL}, y : \tau_2\end{array}}$$