SeLeNe - Self E-Learning Networks
IST-2001-39045

WP2 Deliverable 4.3

Views and Structured Querying in Self e-Learning Networks

Aimilia Magkanaraki, George Kokkinidis, Vassilis Christophides, Gregory Karvounarakis, and Dimitris Plexousakis

Institute of Computer Science, Foundation for Research and Technology, Hellas (FORTH-ICS)

Abstract

Self e-Learning Networks (SeLeNes) aim at facilitating access to digital material — not necessarily primarily produced for educative purposes — to a wide audience of learners and instructors with diverse educational background and requirements. One step towards this goal is the ability to specify educational needs or to describe educational material according to personalized e-learning terminologies and conceptualisations. In this deliverable we investigate how this goal can be achieved in a declarative way using language primitives for defining views over distributed, autonomous RDF bases holding learning object descriptions and schemas. Based on these primitives, we introduce a fully-fledged view definition language, called RVL, for creating not only virtual resource descriptions, but also virtual RDF/S schemas from (meta)classes, properties, and resource descriptions available on a SeLeNe. Furthermore, we illustrate how RVL views can be composed with structured RDF/S queries expressed in a query language like RQL, by means of an internal logical framework (linear Datalog rules) capturing the semantics of the RDF/S model, as well as of RQL queries and RVL views. To the best of our knowledge, RVL is the first declarative view definition language offering such functionality. Finally, we present how the RQL/RVL query processing can be implemented in three different architectural alternatives: the centralized, the mediated and the autonomic scenarios as presented in Deliverable 5.

Heraklion, Crete
January 31, 2004

The SeLeNe Project

Life-long learning and the knowledge economy have brought about the need to support a broad and diverse community of learners throughout their lifetimes. These learners are geographically distributed and highly heterogeneous in their educational backgrounds and learning needs. The number of learning resources available on the Web is continuously increasing, thus indicating the Web's enormous potential as a significant resource of educational material both for learners and instructors.

The SeLeNe Project aims to elaborate new educational metaphors and tools in order to facilitate the formation of learning communities that require world-wide discovery and assimilation of knowledge. To realize this vision, SeLeNe is relying on semantic metadata describing educational material. SeLeNe offers advanced services for the discovery, sharing, and collaborative creation of learning resources, facilitating a syndicated and personalized access to such resources. These resources may be seen as the modern equivalent of textbooks, comprising rich composition structures, "how to read" prerequisite paths, subject indices, and detailed learning objectives.

The SeLeNe Project (IST-2001-39045) is a one-year Accompanying Measure funded by EU FP5, running from 1st November 2002 to 31st October 2003, extended until 31st January 2004. The project falls into action line V.1.9 CPA9 of the IST 2002 Work Programme, and is contributing to the objectives of Information and Knowledge Grids by allowing access to widespread information and knowledge, with e-learning as the test-bed application. The project is conducting a feasibility study of using Semantic Web technology for syndicating knowledge-intensive resources (such as learning objects) and for creating personalized views over such a Knowledge Grid.

Executive summary

This deliverable (4.3) is part of the SeLeNe Workpackage 4 on Syndication and Personalization of Educational Resources. Workpackage 4 has two main objectives:

- To investigate techniques for the syndication and personalization of distributed, autonomous RDF description bases.
- To design language primitives for defining user views over distributed RDF description bases and for deriving composite learning objects' descriptions from those of their constituent learning objects.

Accessing RDF description bases in SeLeNe raises two basic technical challenges: (1) flexible mediation of the different RDF schemas employed by the RDF description bases, and (2) personalization of learning objects' descriptions and schemas according to the educational needs and interests of learning objects' providers (i.e., instructors) and consumers (i.e., learners).

Concerning problem (1), the IEEE LOM has effectively achieved the integration of the various educational metadata standards, as is reported in Deliverable 2.1 of Workpackage 2. Thus, in the context of a SeLeNe, we are assuming that metadata about learning objects are represented using an RDF/S binding of the IEEE LOM. However, fine-grained descriptions expressed in domain or topic-specific taxonomies may also be made available by instructors. Hence, this workpackage is investigating a flexible articulation of different domain/topic-specific taxonomies which can be used for e-learning, as well as the automatic generation of semantic descriptions for composite

learning objects using the descriptions of their constituent learning objects. The taxonomies used for this purpose and the resulting descriptions can be easily represented in RDF/S. This work is reported in Deliverable 4.1.

Concerning problem (2) above, two major issues are involved: (a) specification by learners of their educational needs, and (b) adaptation of learning objects to these needs. The issue (a) requires the representation of educational needs in a "learner profile" using the e-learning schemas, as well as the domain or topic-specific taxonomies available in SeLeNe. It also requires unstructured, keyword-based querying facilities, which can be translated automatically into the structured RDF/S queries supported by the SeLeNe system. The result of these unstructured queries may be returned in a special form of composite learning object called a "trail". To address issue (b) we need methods for dynamically adapting learning material to the preferences of a learner. This requires the ranking of query results by matching the descriptions of the returned learning objects against the learner's profile. These issues are discussed in Deliverable 4.2.

Specifying educational needs or describing educational material according to personalized e-learning RDF/S schemas (for both learners and instructors) requires formalisms for defining declarative views over learning object descriptions and schemas. This work is reported in this deliverable, which also discusses the structured RDF/S querying facilities supported by SeLeNe.

Also needed are techniques for detecting changes in learning objects' descriptions or users' personal profiles, and for notifying users who have subscribed to be notified of such changes. Techniques for the provision of this kind of reactive functionality over RDF descriptions of learning objects and users are reported in Deliverable 4.4.

Revision Information

Revision Date	Version	Changes
August 30, 2003	0.1	First Draft Proposal
September 15, 2003	0.5	Basic Functionality
September 30, 2003	1.0	Formal Model
January 5, 2004	1.5	P2P Query Processing
January 31, 2004	2.0	Final Version (Minor Corrections)

Table of Contents

Contents

1 Introduction **5**

2 A Motivating Example **6**
2.1 Querying Semantic Web Resources with RQL 8
2.2 Defining RVL Views on Semantic Web Resources 10

3 RVL: An RDF View Definition Language **12**
3.1 Design Choices for the RVL View Definition Language 12
3.1.1 Logical Data Independence 13
3.1.2 View Instantiation Capabilities 13
3.1.3 Transformation Expressiveness 13
3.1.4 Closure of View Language 14
3.2 RVL Operators 14
3.2.1 The Instantiation Operator "( )" 14
3.2.2 The Subsumption Operator " <>" 18
3.3 RQL/RVL Internal Logical Framework 19
3.3.1 Translation of RQL Queries into the Internal Logical Framework 20
3.3.2 Translation of RVL Views into the Internal Logical Framework 21
3.3.3 Composing RQL Queries with RVL Views 22

4 RQL/RVL Query Processing in SeLeNe **24**
4.1 Centralized 26
4.2 Mediated 26
4.3 Autonomic 26
4.3.1 RQL Peer Queries 27
4.3.2 Peer Base Advertisement 28
4.3.3 Semantic Query Routing 29
4.3.4 Semantic Query Processing 30
4.3.5 Query Optimization 33
4.3.6 SQPeer Architectural Alternatives 34

5 Related Work **37**
5.1 RVL and View Specification Languages 37
5.2 RQL/RVL Query Processing and Routing 37

6 Summary and Future Work **39**

References **40**

Appendix: RVL Typing Rules **44**

1 Introduction

Personalized access to diverse knowledge bases emerges nowadays as one of the key challenges for Semantic Web [9] applications. For instance, e-learning applications aim to facilitate the access to digital material — not necessarily primarily produced for educative purposes — to a wide audience of learners and instructors with diverse educational backgrounds, needs or expectations. By exploiting the potential of the forthcoming Semantic Web Technology, e-learning applications can effectively support distributed, learner-oriented and non-linear/dynamic learning processes [55].

Self e-Learning Networks (SeLeNes) rely heavily on semantically intensive metadata describing the meaning, usage, accessibility, quality and validity of available educational resources [37]. More and more such metadata is expressed in the **Resource Description Framework/Schema Language** (RDF/S) [41, 15]. Thus, metadata access in a SeLeNe can be easily implemented using a declarative RDF/S query language, such as **RQL** [34]. RQL is a typed, functional query language for uniformly performing navigation and filtering on RDF/S graphs at all abstraction levels (metaschema, schema and data).

However, an RDF/S query language is not enough. As with any query language, formulating queries on complex data may require schema knowledge beyond the needs of a given application. Furthermore, what is useful is not just the retrieval of (part of) an RDF/S graph (as in the case of RQL), but the ability to *create* virtual (meta)schemas and resource descriptions in the result. In fact, for "semantic webs" represented as RDF/S graphs, a view should enable one to restructure the employed class and property hierarchies, as well as to create new resources and class or property types. This gives instructors and learners the ability to access existing learning resources according to their own terminologies and conceptualizations.

The scope of this deliverable is to discuss the language primitives needed for defining declarative views over distributed, autonomous RDF bases holding Learning Object (LO) descriptions and schemas. Based on these primitives, we introduce a fully-fledged view definition language, called RVL, in which one can write views as normal RDF/S schemas and resource descriptions. By exploiting the RQL type system and the distinction of abstraction layers in an RDF/S graph, RVL captures the desired functionality with the use of just two operators. To the best of our knowledge, no language for defining such views has yet been proposed.

Figure 1 shows a SeLeNe where multiple peers are connected, provide LO definitions, pose RQL queries to the system and possibly define RVL views over their local repositories. In this context, a query service should provide a query routing and processing mechanism that considers the intensional (i.e. schema) information for integrating and querying peer bases. Different architectural alternatives for SeLeNe should be considered for the implementation of RQL/RVL query processing, including an autonomic Peer-to-Peer (P2P) scenario.

This deliverable is organized as follows: Section 2 motivates the use of RVL views by means of an e-learning portal example for SeLeNe and exhibits a first sample of the functionality it supports. The design choices which influenced RVL's specification are presented in Section 3, along with the operators it specifies and their respective functionality. The composition of RVL views with structured queries formulated in RQL is exhibited through an internal logical framework, which captures the RDF/S semantics and queries and it is also presented in Section 3. Section 4 analyzes the RQL/RVL query processing mechanism in different architectural alternatives, highlighting the challenges raised by the autonomic scenario, where we propose the ICS-FORTH SQPeer Middleware for Semantic Query Routing and Processing in P2P Database Systems. Sec-

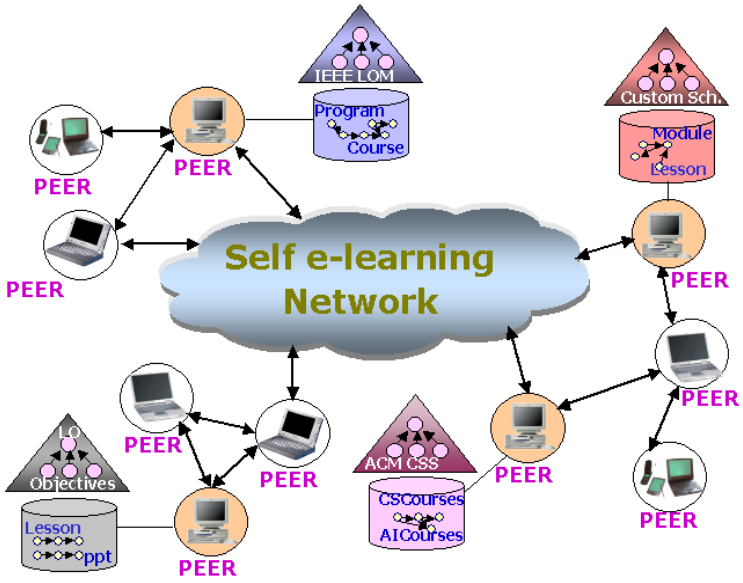

Figure 1: A Self e-Learning Network

tion 5 complements the presentation of RVL and an RQL/RVL query processing and routing mechanism by presenting existing related approaches in the field of relational, object-oriented and XML databases and in the field of P2P query processing and routing. Section 6 concludes by summarizing the contribution of this deliverable and outlining directions for further research.

2 A Motivating Example

We can view each SeLeNe network as a loosely coupled federation of knowledge bases, in which each peer base holds descriptions (i.e., metadata) about the educational material (i.e., LOs) [49] of a learning community. We thus envisage many SeLeNes worldwide, created by and serving different communities of users, as graphically depicted in Figure 1. An instructor can provide and describe new LOs or enrich the descriptions of existing LOs available on the network. Using these descriptions, a learner can retrieve LOs of interest by eventually employing domain or topic-specific taxonomies. In a P2P e-learning network the roles of instructor and learner can of course be played interchangeably. Access to the available LOs is enabled by appropriate *educational portals*, which aggregate and classify in a semantically meaningful way a collection of LOs, which can be used or referenced via a URI [8]. Following the open tradition of the Web, LOs may be physically stored in the web site of an organization (educational or corporate) or in the web pages of individual users. The LO descriptions of a particular learning community may also be physically distributed or stored at the site of the peer hosting that community's portal.

In order to enable effective search for LOs in a SeLeNe, LO descriptions conform to e-learning standards, such as IEEE/LOM (Learning Object Metadata), ARIADNE or IMS[1], as reported in Deliverable 2.1 [56], and also to topic-specific taxonomies of scientific domains, such as ACM/CSS

[1]http://ltsc.ieee.org/wg12/, http://ariadne.unil.ch/Metadata/, www.imsglobal.org/

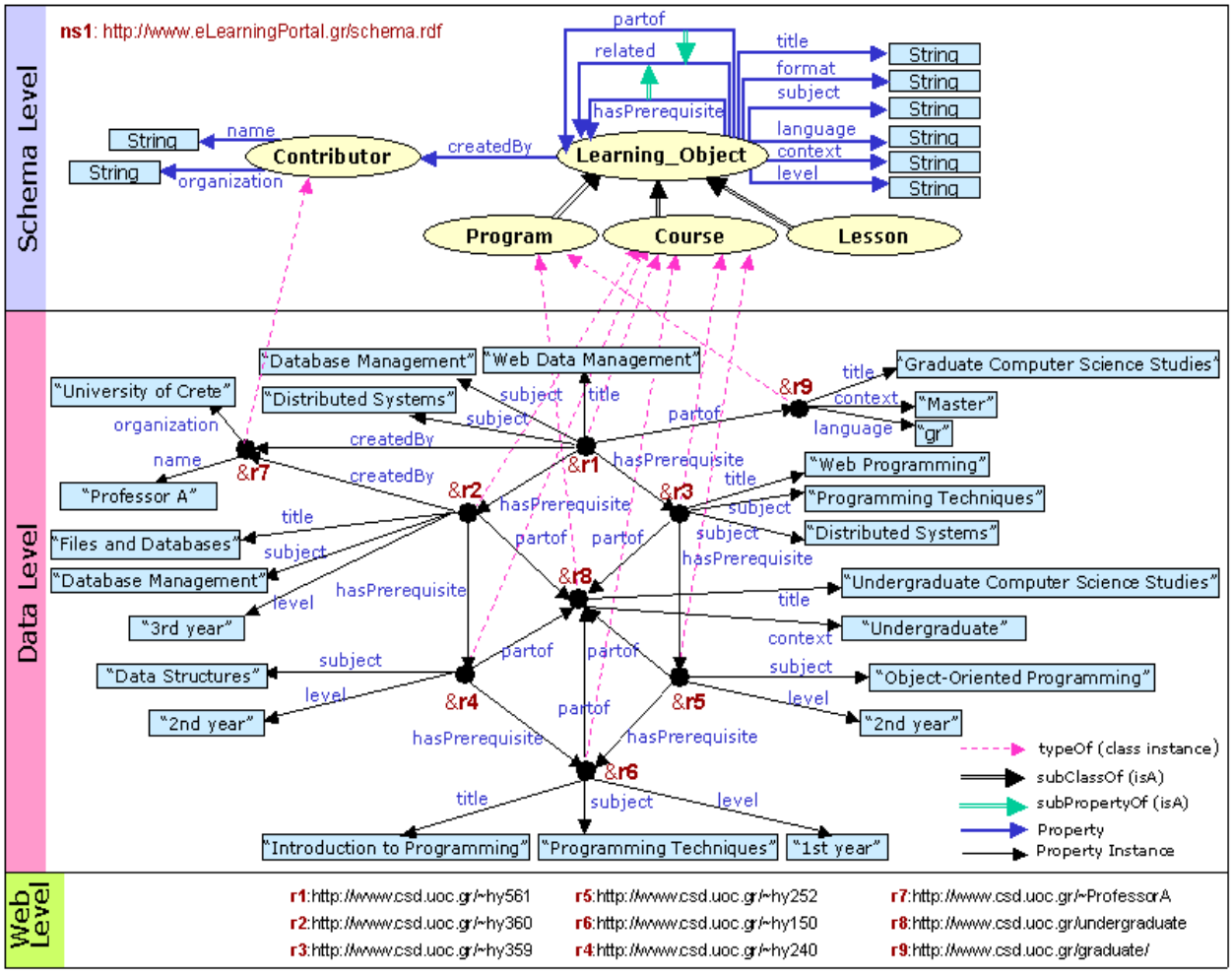

Figure 2: An Example e-Learning Portal Application

(Computing Classification System) [4] or taxonomies of detailed learning objectives [11] [36]. LO schemas and descriptions can be effectively represented in the Resource Description Framework/Schema Language (RDF/S) [41, 15], offering advanced modelling primitives for a SeLeNe information space. In particular, RDF/S (a) enables a *modular design* of descriptive schemas based on the mechanism of namespaces [12]; (b) allows easy *reuse* or *refinement* of existing schemas through *subsumption* of both class and property definitions; (c) supports partial descriptions since *properties* associated with a resource are by default *optional and repeated* and (d) permits *super-imposed descriptions* in the sense that a resource may be multiply classified under several classes from one or several schemas. These modelling primitives are crucial for self e-learning networks, where monolithic e-learning schemas and descriptions cannot be constructed in advance and users may have only incomplete information about LOs.

Figure 2 presents the RDF/S description schema and base of a hypothetical educational portal in SeLeNe. The upper part of the figure depicts a simplified RDF/S schema for describing LOs using attributes with information about their content (*title*, *subject*, *language*, *format*, etc.) and pedagogical value (educational *context* and *level*, learning objectives and time required for

fulfilling an educational unit, etc.). The distinction of the different granularity levels of learning material is represented by the *rdfs:subClass*-es (class subsumption) *Program*, *Course*, *Lesson* or more specific components, such as notes, assignments, exams, figures or simulation programs. Relationships between LOs, like *hasPrerequisite*, capturing learning dependency graphs, or *partof*, capturing learning material composition trees, are defined as *rdfs:subProperty*-s (property subsumption) of the abstract relationship *related*. LOs may also be related to other classes of resources through relationships like *createdBy* ranging over instances of the class *Contributor*, which are described in turn by attributes like *name* and *organization*. These specializations and relationships among LOs are just indicative of the different property types specified in e-learning standards, as reported in Deliverable 2.1 [56], and provide the basic descriptive information needed for effectively querying e-learning information.

The lower part of Figure 2 illustrates how some LOs provided by the web site of the Computer Science Department of the University of Crete (CSD UoC) are described according to the schema employed by the educational portal of our example. For instance, the LO *&r1*, representing the web site of the "Web Data Management" course, is of *rdf:type Course* and has a *title* attribute valued "Web Data Management" and two *subject* attributes valued "Database Management" and "Distributed Systems", respectively. In addition, *Course &r1* is *part of Program &r9* (i.e., the graduate studies *Program* of the CSD UoC), has two *prerequisite* courses *&r2* (with *title* "Files and Database") and *&r3* (with *title* "Web Programming") and it has been *createdBy* the *Contributor &r7* with *name* "Professor A" and *organization* "University of Crete". The other LOs, illustrated in the lower part of Figure 2, are described in a similar way.

2.1 Querying Semantic Web Resources with RQL

Finding LOs in a SeLeNe relies on declarative query languages over RDF/S descriptions, such as RQL [34, 33] (for an exhaustive comparison of RDF/S query languages readers are referred to [42]). RQL offers browsing and querying facilities over the LO information space, i.e., the LO descriptions and the schemas they conform to. More specifically, RQL is a typed query language relying on a functional approach. The type system employed by RQL [34] specifies a set of types, namely the `metaclass of classes`, `metaclass of properties`, `class`, `property`, `resource URIs`, `literal` (XML Schema data types), `bag`, `sequence` and `alternative` types. The kind of restrictions and inferences produced by the use of the RQL type system are presented in [34]. On the whole, these types ensure that RQL, due to its functional nature, can compose basic queries and iterators into complex queries. The basic RQL queries essentially constitute a simple browsing API requiring minimal knowledge of the employed schema(s), while appropriate functions recursively traverse the class/property hierarchies defined in a (meta)schema, such as `subClassOf/subPropertyOf`, and filtering predicates are used to refine the information produced after evaluation.

More generally, placed in a semi-structured context, RQL treats RDF/S description graphs as a collection of nodes and edges. Schema nodes and edges can be queried as normal data using metaclass names, which essentially serve as entry-points to the corresponding graph. Furthermore, RQL supports SQL-like filters, which use generalized path expressions with variables on nodes and edges to traverse RDF/S description graphs at arbitrary depths. Thus, the `SELECT-FROM-WHERE` filters provide a powerful tool to iterate over collections with RDF data or schema information of any kind. The `SELECT` clause, as in SQL, defines a projection over the variables whose values

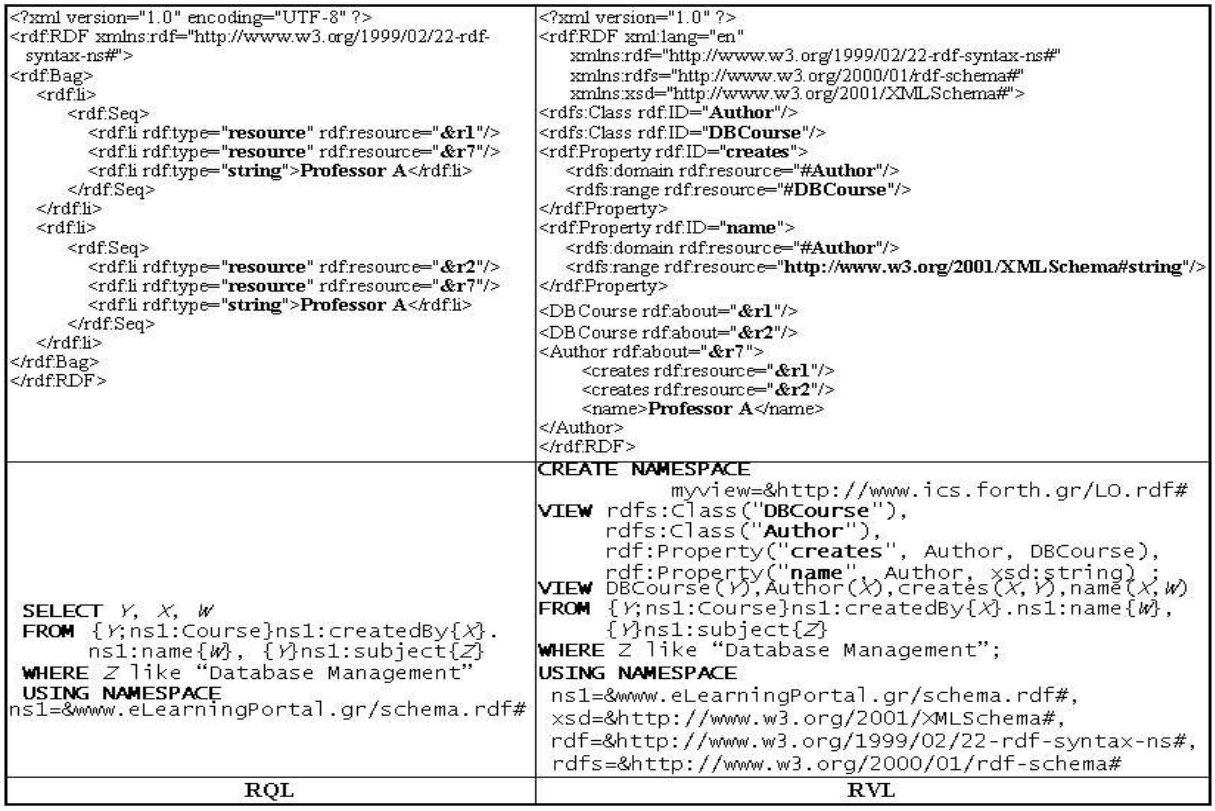

Figure 3: Comparing RQL to RVL

participate in the result. The FROM clause hosts the defined path expressions, which essentially define the part of the RDF/S graph that will participate in the evaluation of the query. In fact, a path expression consists of a series of steps. Each step represents movement in a particular direction by identifying node labels, and each step can apply one or more predicates to eliminate nodes that fail to satisfy a given condition. These filtering conditions are declared at the (optional) WHERE clause. The result of each step is a list of nodes that serves as a starting point for the next step. Moreover, in RDF/S the uniqueness of (meta)schema labels and the ability to describe resources using labels from several schemas is ensured by the XML namespace facility [12]. Thus, RQL uses the (optional) clause USING NAMESPACE for the definition of namespace prefixes.

For instance, consider the RQL query presented on the left side of Figure 3, which retrieves all *Course* resources of *subject* "Database Management" that have been *createdBy* by a resource with a *name* attribute:

```
SELECT Y, X, W
FROM {Y;ns1:Course}ns1:createdBy{X}.ns1:name{W},
     {Y}ns1:subject{Z}
WHERE Z like "Database Management"
USING NAMESPACE ns1=&www.eLearningPortal.gr/schema.rdf#
```

As we see, an RQL FROM clause consists of path expressions, which facilitate the navigation through complex schemas and description bases and bind the introduced variables. Filtering

conditions on these variable bindings are stated in the `WHERE` clause. For instance, the RQL path expression {Y;ns1:Course}ns1:createdBy{X}.ns1:name{W} will match all instances of class *Course* and their associated *createdBy* properties, which link them to some instance of *Contributor* and its *name* value. For each such match, we get a binding that maps Y to the *Course* resource, X to the *Contributor* and W to the *name* value. In a similar way, the path expression {Y}ns1:subject{Z} is evaluated and the variable bindings involved are filtered according to the `WHERE` clause, as well as to the implicit join condition imposed by the presence of the same variable, Y, in both path expressions. The XML serialization of the result of this query is given in the top-left of Figure 3.

2.2 Defining RVL Views on Semantic Web Resources

As well as advanced querying facilities provided by an expressive RDF/S query language such as RQL, also needed is personalization of LO descriptions and schemas employed by a SeLeNe portal. For instance, a learner using an educational portal might want LOs presented according to his/her educational level (e.g., the postgraduate courses) and program (e.g., the e-commerce courses) and not according to the descriptive terms used by the portal designers (e.g., a subject ontology). In this case, the need for a *subjective* rather than an *objective* ontology emerges [24]. An objective ontology (e.g., IEEE LOM) is used by portal designers to describe LOs for quite heterogeneous educational audiences. A subjective ontology models the kind of descriptive terms SeLeNe instructors or learners typically employ to describe a particular domain of interest, thus becoming a classification mechanism of information resources to learning-specific ontologies.

Thus, to enhance the SeLeNe user's experience, we need the ability to personalize the way the portal can be viewed, by providing simpler virtual schemas that reflect an instructor's or learner's perception of the domain of interest. *RVL*, the view definition language considered in this deliverable, provides this ability by offering techniques, on the one hand, for the reconciliation and integration of metadata describing heterogeneous distributed LOs and, on the other hand, for the definition of personalized views over the LO information space.

To illustrate the functionality of RVL, consider a simple virtual schema (view) for instructors, which represents only database course material and its authors. This schema can be specified with the RVL statements presented in the bottom-right part of Figure 3 taking as input the RDF/S description base of Figure 2. The output of these view statements is the RDF/S virtual schema and resource descriptions presented at the top-right part of Figure 3 in an XML serialization.

Since in RDF/S the uniqueness of (meta)schema labels and the ability to describe resources using labels from several schemas is ensured by the XML namespace facility [12], in our example we use the RVL statement:

```
CREATE NAMESPACE myview=&http://www.ics.forth.gr/LO.rdf#
```

Descriptive labels are prefixed by the namespace of the schema to which they belong (e.g., *ns1#Learning_Object*), forming in this way unique URIs. This is particularly important in the open and diverse Web world and even more so when defining views, where virtual, but different, copies of old schema labels, such as class and property names, are considered.

The second RVL statement in our example "creates" the virtual classes `Author` and `DBCourse` and the virtual properties `creates` and `name`:

```
VIEW rdfs:Class("DBCourse"),rdfs:Class("Author"),
     rdf:Property("creates", Author, DBCourse),
```

```
rdf:Property("name", Author, xsd:string);
```

where *rdfs:Class* and *rdf:Property* are two core metaclasses provided in the default RDF/S namespaces. The semantics of these namespaces along with the XML Schema datatypes is built-in in RVL/RQL and the corresponding namespace prefixes (e.g., *rdf*, *rdfs*,*xsd*) can thus be omitted[2]. Furthermore, we can use the `USING NAMESPACE` clause to declare the namespaces used in view statements. As we will see in the next section, RVL also provides the ability to create virtual subsumption hierarchies or even to filter/restructure existing ones.

The third RVL statement "populates" the virtual classes and properties defined in the view with appropriate instances copied from the source description base illustrated in Figure 2:

```
VIEW DBCourse(Y),Author(X),creates(X,Y),name(X,W)
FROM {Y;ns1:Course}ns1:createdBy{X}.ns1:name{W},
     {Y}ns1:subject{Z}
WHERE Z like "Database Management";
```

This statement works much like the query on the portal description base presented in Section 2.1. To emphasize the connection, we juxtapose on Figure 3 the RVL statement and the RQL query having the same `FROM` and `WHERE` clauses. In the top row of Figure 3, we give the XML serialization of the results of both the query and the view definition statements.

As we can observe, an RVL `FROM` clause consists of the RQL [33] path expressions facilitating navigation through complex schemas and description bases and binding of introduced variables. Filtering conditions on these variable bindings are stated in the `WHERE` clause, as in RQL queries. Notice however the difference between the result of the RQL query and the output of the RVL view definition in Figure 3. Although their input is the same RDF/S graph, RVL is capable of producing virtual schemas and resource descriptions instead of simple variable bindings represented in some (nested) tabular form.

This functionality is ensured by the `VIEW` clause, where appropriate population functions are used taking as parameters the variable bindings produced by the `FROM-WHERE` filter. For instance, the virtual class `DBCourse` is populated with instances (bound to variable Y) of the original class *Course* having a property *subject* valued "Database Management". The virtual class `Author` is populated with instances (bound to variable X) of the base class *Contributor*, which are the range values of the property *createdBy* applied to *Course* resources. In other words, `Author` is populated with all the contributors who have created a database course. Virtual properties are populated with pairs of resources (e.g., `creates` is populated with authors having created database courses) or pairs of resources-values (e.g., `name` is populated with the names of database course authors). One of the most salient RVL features is its ability to create virtual schemas by simply populating the two core RDF/S metaclasses *Class* (e.g., with schema classes `Author` and `DBCourse`) and *Property* (e.g., with schema properties `creates` and `name`). For someone interested only in database learning material, this view is much easier to understand. One can then easily formulate queries *on the view*, such as the following one in RQL:

```
SELECT Y
FROM {X}myview:creates{Y}, {X}myview:name{Z}
WHERE Z = "Professor A"
USING NAMESPACE myview=&http://www.ics.forth.gr/LO.rdf#
```

[2]For illustration purposes, the default namespaces are included in the example.

This query should retrieve the database courses created by the author named "Professor A".

In the context of a SeLeNe, a view definition language such as RVL can be used to implement advanced user aids, such as personalized *navigation* and *knowledge maps*. In the first case, using appropriate GUIs implemented on top of SeLeNe application services [52], one can easily implement 'smart' bookmarks on available LOs, whose access implies a sequence of navigation steps and/or complex filtering conditions on the underlying ontology (or ontologies) of an educational portal. To use this bookmark and access its relevant LOs we just need to activate the corresponding view. Moreover, a SeLeNe user can further describe a smart bookmark using properties from another schema such as the educational objective of the contained LOs, etc. In the second case, knowledge maps define and combine neighbouring semantic domains for SeLeNe sub-communities. Such knowledge maps essentially enable SeLeNe 'super-peers' [52] to create personalized semantic portals by clustering in a meaningful way peer description bases relevant to the educational needs and interests of an e-learning sub-community.

In the following sections we detail the mechanisms that RVL utilizes to accomplish the previously described functionality.

3 RVL: An RDF View Definition Language

Motivated by the previous example, a fundamental question one can naturally pose, is "*what is a good specification of views for the RDF/S data model?*". We have designed RVL as a conceptually simple language enabling both humans and applications to understand view specifications as normal RDF/S schemas and resource descriptions. More precisely, an RVL view specifies a **virtual description schema** graph (or **virtual schema** for brevity). Its extension corresponds to a **virtual description base** graph (or **virtual base** for brevity), which is a valid instance of the virtual view schema. Thus, RVL views produce new RDF/S (meta)classes and properties which are **virtual** and their instances are computed from the **source** base(s) or schema(s) using the RVL program specifying the view. This program defines essentially the *mapping* (i.e., transformation) of the input (i.e., source) to the output (i.e., virtual) RDF/S graph(s).

3.1 Design Choices for the RVL View Definition Language

In order to design an effective RDF/S view specification language, we have addressed the following issues:

1. How are the virtual schema (meta)classes and properties of a view related to the source description schema(s)?
2. How are the virtual base resources and property values of a view related to source description base(s)?
3. What is the expressiveness of the input/output transformations supported by the view specification language?
4. How can the output of view specifications be used in queries and other views?

In the following subsections, we will present the main design choices for RVL in response to the above fundamental issues.

3.1.1 Logical Data Independence

Logical data independence is one of the most important properties that a view definition language should respect (recall the ANSI-SPARC three-level architecture [6]). It essentially requires that view specifications should be independent of those of the source schemas and bases, while the semantics of existing virtual schemas should not be altered by the definition of new ones. For this reason, the scope of virtual (meta)class and property definitions is determined in RVL by the namespace of the view. This is particularly useful since RVL allows us to not only create *new* (meta)classes and properties (as in Figure 3), but to also *import* in a view existing ones from the source schemas given as input. Imported (meta)classes and properties are simply replicated in the virtual schema and do not interfere with those at the source. Moreover, as we will see in Section 3.2, virtual subsumption hierarchies (for both classes and properties) could also be defined in a view, which are not necessarily present in the source schemas. Instead of creating a global subsumption hierarchy mixing both virtual and source (meta)classes and properties, an RVL virtual schema refers only to the subsumption relationships explicitly established between the virtual (meta)classes and properties. The separation of virtual from source (meta)classes and properties in RVL leads to smaller virtual schemas easier to understand and manage.

3.1.2 View Instantiation Capabilities

Besides the population of virtual (meta)classes and properties using, for instance, RQL queries (see Figure 3) over the original description base (i.e., **object-preserving views**), an RVL virtual schema can also be instantiated in the view (i.e., **object-generating views**) specification. These instances exist only during the activation of the view and their identifiers are generated by appropriate Skolem functions. As a matter of fact, the entire virtual schema specified in a view is essentially a new instance of the default RDF/S meta-schema (class and property names are used as unique identifiers)! As we will see in Section 3.2, this functionality is also useful in cases where virtual resource descriptions may have both a dynamic part populated with resources from the original base and a static one populated exclusively at the view level. RVL is powerful enough to support both kinds of view instantiation, while instances of the source schemas are simply copied into the view extension, thereby acquiring a virtual hypostasis.

3.1.3 Transformation Expressiveness

Transformation expressiveness is the cornerstone of the RVL design in order to cope with a wide range of heterogeneities found in real-scale Semantic Web applications [39, 10]. Therefore, a view specification language should provide the ability to both create (for personalization purposes) and reconcile (for mediation purposes) quite different conceptual representations of the same application domain. For this reason, RVL is equipped with *expressive restructuring* capabilities enabling users to change the abstraction level (i.e., metaschema, schema, data) in which a particular view construct is defined. As we will detail in Section 3.2, RVL is capable of "promoting" literals or resources of the original description base to virtual classes, as well as of "demoting" metaclasses of the original description schema to virtual classes of the view. This ability is ensured by the expressiveness of the RQL query language to query RDF/S information at all abstraction levels and the polymorphic type system of the RVL population functions (i.e., the `VIEW` clause) (see Table 1 in the Appendix).

3.1.4 Closure of View Language

On the one hand, one should be able to query RVL views, as in the case of source schemas and description bases. Since RVL views introduce virtual schemas, one can use their namespace to formulate RQL queries retrieving (part of) the RDF/S graph specified by the view program. On the other hand, one should be able to create views using both source and virtual schemas. We can distinguish between two levels of view specification reuse: inside a virtual schema (intra-) and across (inter-) virtual schemas. Intra-view reuse is not supported by RVL, since it gives the possibility to define the extension of a virtual (meta)class based on the extension of another virtual construct of the same view. To ensure data independence and avoid cyclic declarations of virtual classes which are hard to grasp, we impose the following restriction: the `FROM` clause of RQL queries defining the population of the view constructs cannot refer to information (schema and data) of the view being defined. Only inter-view reuse is supported by RVL for creating virtual (meta)classes and properties by employing other virtual schemas. This process results in a cascade of virtual schema specifications, which ensures that the constructs of a virtual schema used in the definition of another virtual schema have already been defined.

The above design decisions were taken with the objective of devising a clear and expressive RDF/S view specification language required by a large spectrum of Semantic Web applications. In the following subsections, we will detail how RVL implements this functionality.

3.2 RVL Operators

RVL reduces the creation of virtual schemas and description bases down to the use of two operators, namely the **instantiation** and the **subsumption** operators. In order to ensure the validity of their application and infer the type of virtual constructs, the operands of the RVL operators must be of a specific type, which is checked during compilation with respect to the RQL/RVL type system using the typing rules presented in Table 1 in the Appendix. In addition, the presence of this type system, facilitates a more compact declaration of view statements, in the sense that the type of one entity in the source schema or base can be reused as such in the view. This ability does not prohibit users from altering the type of one element using the instantiation operator, as we will subsequently see in this section.

In the following, we will sketch the functionality supported by each operator by using the more complex view illustrated in Figure 4. This virtual schema is defined as a view on the schema of the motivating example in Figure 2 and refers to computer science courses — especially database and programming language courses — and their authors. In each case, we cite the typing rule of Table 1 (in the Appendix) applicable for the specific operator.

3.2.1 The Instantiation Operator "( )"

The instantiation operator, denoted "`()`", exploits the existence of abstraction layers in an RDF/S graph in order to support: (a) the creation/import of virtual (meta)classes and properties and (b) the population of virtual (meta)classes and properties. The instantiation of a virtual construct should be performed only with resources at the immediate lower abstraction level (see rules `9-12` in Table 1 in the Appendix). Changing the type of an RDF/S entity in an RVL view compared to a source schema or base (e.g., a literal to a class, or a metaclass to a class) is also supported using more complex RVL expressions.

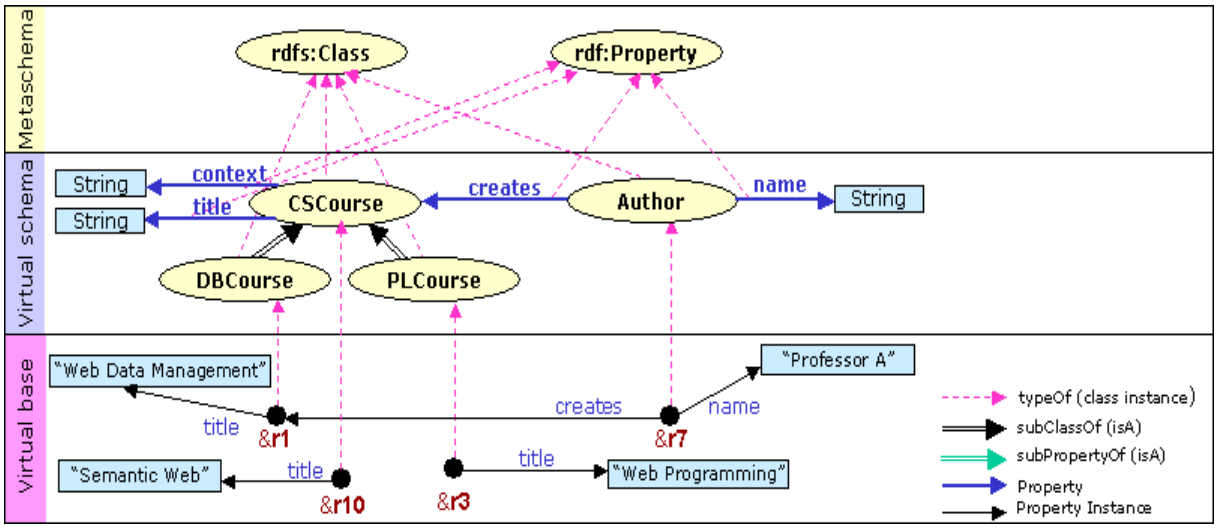

Figure 4: A More Complex RVL View

Let us examine the functionality of the instantiation operator by means of the example view illustrated in Figure 4. In the simplest case, we are interested in creating new virtual classes as follows (see rule 3):

```
VIEW
Class("Author"),Class("CSCourse"),Class("DBCourse"),Class("PLCourse");
```

The first operand of "()" is the (meta)class (e.g., *Class*) one wants to populate with a new instance identified by the string value of the second operand (e.g., `Author`). Virtual metaclasses of classes and properties can be also created by instantiating the RVL built-in (meta)metaclasses *rvl:MetaClass* and *rvl:MetaProperty* (see rules 1 and 2 respectively). In order to import a part (i.e., a *set*) of the classes defined in a source schema, we need first to use an RQL filter in order to identify which classes (or properties) are going to be imported into the virtual schema and then, use the instantiation operator in the `VIEW` clause, as depicted by the following example (see rule 3):

```
VIEW Class(X)
FROM Class{X}
WHERE namespace(X) = ns1 and X < ns1:Learning_Object;
```

The RQL path expression *Class*$\{X\}$ in the `FROM` clause introduces a variable X ranging over all classes, while the `WHERE` clause filters X bindings only to the subclasses (direct or transitive) of *Learning_Object* defined in the schema namespace *ns*1. The instantiation operator "()" in the `VIEW` clause simply creates new instances of *Class* for each successful binding of class variable X. Since in this case we are importing in the virtual schema classes as provided by the source schema, we can omit the explicit call to the instantiation operator by just writing `VIEW X`.

This abbreviation cannot be used when we transform ("promote" or "demote") the abstraction level (i.e., metaschema, schema, data) of constructs specified in the view with respect to their level in the source schema and base. Assuming, for instance, that the values of the property *subject* are not simple strings but terms from a structured vocabulary (e.g., ACM Computing

Classification System[3]), one can easily create virtual classes from these values using the following RVL statement (see rule 3):

```
VIEW Class(X)
FROM ns1:subject{X};
```

In this example, string values will be used as unique names of the so created virtual classes. For this purpose, the instantiation operator uses appropriate Skolem functions: for two equal *subject* values, only one virtual class is created. This ability offers a great flexibility in view specification, especially in environments, such as self e-learning, with highly diverse modelling of resource descriptions.

As far as **properties** are concerned, RVL follows the RDF/S approach to consider properties as first-class citizens. Thereby, their definition is independent of the definition of the class they are attributed to, while they can be specialized forming subsumption hierarchies. The restriction posed by the RQL/RVL data model is that the domain and range of a property must always be defined and be unique, thus the creation of a (virtual) property is accompanied with the definition of its domain and range classes (or metaclasses or literal types). To accommodate this peculiarity, the instantiation operator has a slightly different syntax. The first operand of the instantiation operator corresponds to the name of the core metaclass of properties (*Property*), the second to the name of the virtual property, the third to its domain and the fourth to its range. In the simplest case, we are interested in creating new virtual properties as follows (see rule 4):

```
VIEW Property("creates", Author, CSCourse),
     Property("name", Author, xsd:string),
     Property("context", CSCourse, xsd:string),
     Property("title", CSCourse, xsd:string);
```

This view statement creates four new instances of the metaclass *Property* uniquely identified by their names: the virtual property `creates` emanating from the virtual class `Author` and ranging over the virtual class `CSCourse`, as well as the virtual attributes `name`, `context` and `title` of type `string` having as domain respectively the virtual classes `Author` and `CSCourse`.

Due to the functional nature of RVL, the operands of the instantiation operator are not restricted to atoms (constants or variables), but can also be other RVL/RQL expressions of an appropriate type. For instance, we could define *inverse* properties using the following RVL statement (see rule 4):

```
VIEW Property("creator",range(ns1:createdBy),domain(ns1:createdBy));
```

In this example, the virtual property `creator` is created with domain and range the virtual classes `Contributor` and `Learning_Object` respectively returned by the employed RQL functions. This is an example of another possible RVL abbreviated expression: the domain and range virtual classes `Contributor` and `Learning_Object` are defined in the view at the same time as the property `creator`. The complete syntax of the `VIEW` clause comprises the expressions: `Class(domain(ns1:createdBy))` and `Class(range(ns1:createdBy))`.

As in the case of classes, we can import in the view a part (i.e., a *set*) of the properties defined in a source schema as follows (rule 4):

```
VIEW Property(P, CSCourse, range(P))
FROM Property{P}
WHERE domain(P)=ns1:Learning_Object and P < ns1:related;
```

[3]http://www.acm.org/class/1998/

According to our example of Figure 2, this RVL statement creates two instances of the metaclass *Property* with names `partof` and `hasPrerequisite` with domain the already defined virtual class `CSCourse` and with the same ranges as in the source schema identified by the namespace `ns1`.

Besides creating virtual schemas we also need to populate the virtual classes and properties specified in the view. The same instantiation operator is used for this purpose, this time taking operands of different types. The additional restriction imposed in the case of properties is that the resources at the data level to which a property is attributed are instances of the domain and range classes of the property at schema level. The following two RVL statements populate the virtual classes and properties we defined above for the example of Figure 4 (see rules `11` and `12` respectively):

```
VIEW DBCourse(Y),creates(X,Y),Author(X),name(X,W),context(Y,Z),title(Y,K)
FROM {Y;ns1:Course}ns1:createdBy{X}.ns1:name{W}, {Y}ns1:context{Z},
     {Y}ns1:title{K}, {Y}ns1:subject{L}
WHERE L like "Database Management";
VIEW PLCourse(Y),creates(X,Y),Author(X),name(X,W),context(Y,Z),title(Y,K)
FROM {Y;ns1:Course}ns1:createdBy{X}.ns1:name{W}, {Y}ns1:context{Z},
     {Y}ns1:title{K}, {Y}ns1:subject{L}
WHERE L like "*Programming*";
```

The virtual class `DBCourse` (`PLCourse`) is populated with instances of the source class *Course* having a property *subject* valued "Database Management" ("Programming Techniques" or "Object-Oriented Programming"). The virtual class `Author` is populated in both cases by *Contributor* instances having created (property *createdBy*) *Course* instances on the desired *subject*. Virtual properties are populated in a similar way (`DBCourse` and `PLCourse` are defined as subclasses of `CSCourse` in the next section).

As a last example, we illustrate how virtual classes (or properties) can be populated with virtual resources residing exclusively at the view. Assuming that an instructor wants also to include within the virtual base `CSCourse`s a self-published course on the "Semantic Web", the following RVL statement can be issued through a suitable, easy-to-use interface (provided by SeLeNe's Presentation service), which automatically generates the appropriate RQL/RVL statements (rules `11` and `12`):

```
VIEW CSCourse(&http://www.mycourses.net/~SemWeb),
     title(&http://www.mycourses.net/~SemWeb, "Semantic Web");
```

As we will see in the next subsection, by defining `DBCourse` and `PLCourse` as subclasses of `CSCourse`, the final population of `CSCourse` will contain its proper instances, as well as those of its subclasses.

In more complex situations, an instructor may want to populate the `DBCourse` virtual class with resources from a source base, while completing their description manually, by adding, for instance, a learning `objective` property:

```
VIEW DBCourse(X),objective(X,"research tutorial")
FROM {X;ns1:Course}ns1:subject{Y},
WHERE Y like "Database Management";
```

The above RVL statement will create for each LO instance of `DBCourse` an `objective` property with value "research tutorial" (the property is assumed to have already been defined in the view).

3.2.2 The Subsumption Operator " <>"

The subsumption operator, denoted "`< >`", is mainly used for defining virtual sub-(meta)classes or subproperties. Some restrictions are imposed on the use of this operator by the *RQL/RVL* data model. First, cycles in virtual class (or property) subsumption hierarchies are not allowed. Second, the domain and range of a property must be subsumed by the domain and range of its super properties. In addition, the subsumption operator is applicable on operands of the same type ((meta)/class and property types), since the formulation of hierarchies between entities of different type is meaningless (see rules `5-8` in Table 1 in the Appendix).

In the simplest case, one wants to explicitly define the subsumption relationship between two virtual (meta)classes or properties, as for instance in the following RVL statements:

```
VIEW CSCourse<DBCourse>;
VIEW CSCourse<PLCourse>;
```

The second operand (e.g., `DBCourse`) of "`< >`" is declared to be a subclass (or a subproperty) of the first one (e.g., `CSCourse`). Both operands in this example are of type class (see rule `7`).

As we have seen in the previous subsection, RVL gives us the ability to import a part of the source schema into the view. Using the subsumption operator in conjunction with RQL filters, we are able to import not only the source classes (or property) names, but entire subsumption hierarchies from a source schema, as depicted in the following example:

```
VIEW $X<$Y>
FROM $X{;$Y}
WHERE namespace($X)=&www.eLearningPortal.gr/schema.rdf# and
      namespace($Y)=&www.eLearningPortal.gr/schema.rdf#;
```

The RQL path expression in the `FROM` clause essentially traverses the class subsumption hierarchy of the source schema identified by the namespace `www.eLearningPortal.gr/schema.rdf`. Then, for each binding of the class variable $\$X$ (e.g., to *Learning_Object*), the variable $\$Y$ is bound to the corresponding (direct or transitive) subclasses (e.g., to *Course*). The result of the original RQL query essentially produces a Cartesian product of each class with its subclasses. The use of the subsumption operator in the `VIEW` clause, with the variables $\$X$ and $\$Y$ as operands, results in the reconstruction of the original subsumption hierarchy of the source schema in the view. It should be stressed that the above RQL path expression considers a complete transitive closure of the subsumption hierarchy (i.e., all the paths from a node to its ancestors up to the root exist). This is extremely useful when filtering conditions on class (or property) names are also used in the `WHERE` clause. For instance, the exclusion from the view of some source classes (e.g., *Program*) results in a "connected" hierarchy related through subsumption subclasses (e.g., figures, exams, etc.) to their ancestors(s) (e.g., *Learning_Object*). Since the use of appropriate labelling schemes for class (or property) Directed Acyclic Graphs (DAGs) [19] alleviates the need for actually computing the transitive closure, the subsumption operator can easily produce a *minimal* form in which redundant relationships are removed.

The RVL examples presented in this section were just indicative of RVL's expressiveness. Consider the spectrum of possible views which can be defined by changing the operands of the subsumption and instantiation operators and by exploiting the querying capabilities of RQL. This expressiveness allows us to think of RVL as a powerful transformation mechanism for RDF/S schema and resource description graphs. In addition, RVL allows the capture of several modelling constructs recently proposed in OWL [22], such as inverse properties, synonyms of classes

and properties or complex class definitions using boolean expressions and existential/universal quantifiers (supported by RQL filters) in a view.

3.3 RQL/RVL Internal Logical Framework

In order to fully capture the semantics of RQL queries and RVL views and to formally ground the ability to compose RVL views with RQL queries, we use an internal logical framework, which captures the semantics of RDF/S schemas and queries/views [18]. This first-order, relational framework is used to translate both queries and views into a common formalism providing methods to ensure the validity of logical operations. More specifically, this internal logical framework employs first-order relations together with some first-order constraints to model RDF/S and uses a signature with three sorts: $\mathsf{Resource}, \mathsf{Property}, \mathsf{Class}$[4]. The relations used, along with their respective meaning, are listed below:

- $\mathsf{C\_EXT}(c, x)$ iff the resource x is in the *proper extent* (i.e., it is a direct (proper) instance) of class c. According to the RDF specification [41], class extents can overlap due to multiple classification of resources;
- $\mathsf{C\_SUB}(c, d)$ iff c is a (not necessarily direct) subclass of d;
- $\mathsf{PROP}(c, p, d)$ iff class c is the domain and class d is the range of property p;
- $\mathsf{P\_EXT}(x, p, y)$ iff (x, y) is in the *proper extent* (i.e., it is a direct (proper) instance) of property p [5]. Like class extents, property extents may overlap;
- $\mathsf{P\_SUB}(p, q)$ iff p is a (not necessarily direct) subproperty of q.

In order to be compliant with the RDF/S semantics, these relations must satisfy some *built-in* constraints imposed by the RQL/RVL data model. In particular, the domain and range of a property must be unique, while the subclass and subproperty relations must be reflexive, transitive and satisfy the following *subproperty/subclass compatibility* constraint:

$$\forall\, a, p, b, c, q, d\ \mathsf{P\_SUB}(q, p) \ \wedge\ \mathsf{PROP}(a, p, b) \ \wedge\ \mathsf{PROP}(c, q, d) \ \longrightarrow\ \mathsf{C\_SUB}(c, a) \ \wedge\ \mathsf{C\_SUB}(d, b)$$

This means that if q is a subproperty of p, the domain and range of q are subclasses of the domain and range of p, respectively.

Furthermore, we have the *property-class extent compatibility* constraint, i.e., any instance of a property p connects a pair of instances of some subclasses of the domain and range of p, respectively:

$$\forall\, a, p, b, x, y\ \mathsf{PROP}(a, p, b) \ \wedge\ \mathsf{P\_EXT}(x, p, y)$$
$$\longrightarrow\ \exists\, c, d\ \mathsf{C\_SUB}(c, a) \ \wedge\ \mathsf{C\_SUB}(d, b) \ \wedge\ \mathsf{C\_EXT}(c, x) \ \wedge\ \mathsf{C\_EXT}(d, y)$$

If we denote with Δ_{RDF} the set of dependencies (constraints) used to axiomatize the internal RDF/S model, then the following theorem holds:

[4]For simplicity reasons, we ignore metaclasses and metaproperties in this discussion, but they can be handled easily in the same way.

[5]In our model, instances of properties are represented as ordered pairs of the resources they connect.

Theorem 1 *It is decidable whether* $\Delta_{\text{RDF}} \models d$ *and whether* $\Delta_{\text{RDF}} \models Q_1 \sqsubseteq Q_2$*, where* d *is an embedded implicational dependency,* Q_1, Q_2 *are conjunctive queries and* $\sqsubseteq$ *is query containment.*

Using the relations C_SUB, PROP and P_SUB and the names of classes and properties defined in a schema as constants, we can straightforwardly translate the information embodied into an RDF/S schema to this internal framework as a set of relational facts (in Datalog parlance, an extensional database). For instance, some of the facts obtained from the schema of Figure 2 are:

C_SUB(Course, Learning_Object)
PROP(Learning_Object, createdBy, Contributor)
P_SUB(partof, related)

Note that this set of facts will include all C_SUB and P_SUB reflexivity instances and will be "closed" under transitivity and under subproperty/subclass compatibility.

3.3.1 Translation of RQL Queries into the Internal Logical Framework

An RQL *conjunctive* query is a query of the form $ans(\bar{X}) :- C_1, \ldots, C_n$ where the C_i's are either RQL class or property patterns (as they appear in the RQL FROM clause) or equalities involving variables and/or constants and $\bar{X}$ is a tuple of variables or constants (range restrictions [1] are also required). Many RQL queries are in fact conjunctive queries, e.g., the query in Section 2 can be written [6]:

```
ans(Y, X, W) :- {Y;Course}createdBy{X}, {X}name{W},
         {Y}subject{Z}, Z = "Database Management"
```

Conjunctive RQL queries can then be translated into relational conjunctive queries in the internal logical framework. Indeed, according to the declarative semantics in [34], RQL patterns have the same meaning as conjunctions of relational atoms. For example:

RQL Pattern	*Internal Model Translation*
$\{X;\$C\}@P\{Y;\$D\}$	$\mathsf{PROP}(a,p,b)$, $\mathsf{P\_SUB}(q,p)$, $\mathsf{P\_EXT}(x,q,y)$, $\mathsf{C\_SUB}(c,a)$, $\mathsf{C\_SUB}(d,b)$, $\mathsf{C\_EXT}(c,x)$, $\mathsf{C\_EXT}(d,y)$
$\{X;\$C\}@P\{Y\}$	$\mathsf{PROP}(a,p,b)$, $\mathsf{P\_SUB}(q,p)$, $\mathsf{P\_EXT}(x,q,y)$, $\mathsf{C\_SUB}(c,a)$, $\mathsf{C\_EXT}(c,x)$
$\{X\}@P\{Y\}$	$\mathsf{P\_SUB}(q,p)$, $\mathsf{P\_EXT}(x,q,y)$

In the above RQL patterns, X, Y are resource variables, $\$C, \$D$ are class variables (and can be replaced with constant class names), and $@P$ is a property variable (that can also be replaced by a constant property name). Using these patterns, the RQL conjunctive query above translates internally to the following Datalog rule:

[6] Namespaces are omitted for clarity reasons. Furthermore, for demonstration reasons, the "like" condition defined in the WHERE clause of the example query is replaced with equality.

```
ans(y,x,w) :- PROP(a,createdBy,b),P_SUB(q1,createdBy),P_EXT(y,q1,x),
              C_SUB(Course,a),C_EXT(Course,y),
              P_SUB(q2,subject),P_EXT(y,q2,z),z="Database Management"
```

3.3.2 Translation of RVL Views into the Internal Logical Framework

In order to favour personalization, virtual RDF/S schemas can be also specified on top of the portal schema, as for instance the RVL schema shown in Figure 4. If we restrict our attention to "conjunctive" RVL definitions, virtual classes' and properties' definitions and extents can also be written as rules, whose heads are as in the `VIEW` clause in RVL and bodies are just like conjunctive RQL queries. For instance:

```
Class(DBCourse) :-
Class(PLCourse) :-
Class(CSCourse) :-
CSCourse<DBCourse> :-
CSCourse<PLCourse> :-
Class(Author) :-
Property(creates, Author, CSCourse) :-
Property(name, Author,String) :-

DBCourse(Y):- {Y;Course}subject{Z}, Z="Database Management"
PLCourse(Y):- {Y;Course}subject{Z}, Z="Programming"
Author(X):-   {Y;Course}createdBy{X}, {Y}subject{Z}, Z="Database Management"
Author(X):-   {Y;Course}createdBy{X}, {Y}subject{Z}, Z="Programming"
creates(X,Y):-{Y;Course}createdBy{X}, {Y}subject{Z}, Z="Database Management"
creates(X,Y):-{Y;Course}createdBy{X}, {Y}subject{Z}, Z="Programming"
name(X,W):-   {Y;Course}createdBy{X}, {Y}subject{Z}, Z="Database Management",
              {X}name{W}
name(X,W):-   {Y;Course}createdBy{X}, {Y}subject{Z}, Z="Programming", {X}name{W}
```

All the above rules can be translated into a set of conjunctive relational views. To accomplish this translation, the internal framework is equipped with relations similar to those presented in Section 3.3. This allows the capture of virtual classes and properties, as well as their virtual subsumption relationships, namely CLASS_V, PROP_V, C_EXT_V, P_EXT_V, C_SUB_V and P_SUB_V, respectively. Thus, for the rules presented previously, the conjunctive relational views created are:

```
CLASS_V(DBCourse) :-
CLASS_V(PLCourse) :-
CLASS_V(CSCourse) :-
C_SUB_V(DBCourse,CSCourse) :-
C_SUB_V(PLCourse,CSCourse) :-
CLASS_V(Author) :-
PROP_V(Author, creates, CSCourse) :-
PROP_V(Author, name, String) :-
```

```
C_EXT_V(DBCourse,y) :-  PROP(a,subject,b),P_SUB(q,subject),P_EXT(y,q,z),
                        C_SUB(Course,a),C_EXT(Course,y),z="Database Management"
C_EXT_V(PLCourse,y) :-  PROP(a,subject,b),P_SUB(q,subject),P_EXT(y,q,z),
                        C_SUB(Course,a),C_EXT(Course,y),z="Programming"
C_EXT_V(Author,x) :-    PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
                        C_SUB(Course,a1),C_EXT(Course,y),
                        P_SUB(q2,subject),P_EXT(y,q2,z),z="Database Management"
C_EXT_V(Author,x) :-    PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
                        C_SUB(Course,a1),C_EXT(Course,y),
                        P_SUB(q2,subject),P_EXT(y,q2,z),z="Programming"
P_EXT_V(x,creates,y) :- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
                        C_SUB(Course,a1),C_EXT(Course,y),
                        P_SUB(q2,subject),P_EXT(y,q2,z),z="Database Management"
P_EXT_V(x,creates,y) :- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
                        C_SUB(Course,a1),C_EXT(Course,y),
                        P_SUB(q2,subject),P_EXT(y,q2,z),z="Programming"
P_EXT(x,name,w) :-      PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
                        C_SUB(Course,a1),C_EXT(Course,y),
                        P_SUB(q2,subject),P_EXT(y,q2,z),z="Database Management",
                        P_SUB(q3,name),P_EXT(x,q3,w)
P_EXT(x,name,w) :-      PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
                        C_SUB(Course,a1),C_EXT(Course,y),
                        P_SUB(q2,subject),P_EXT(y,q2,z),z="Programming",
                        P_SUB(q3,name),P_EXT(x,q3,w)
```

Note that the schema part of these conjunctive relational views needs to be "completed", i.e., closed, under reflexivity and transitivity.

3.3.3 Composing RQL Queries with RVL Views

Having set up a logical framework that captures the RDF/S semantics, we then translated both RQL queries and RVL views into this framework. The objective of this translation is to provide a logical framework into which queries formulated against a view are combined with the definition of the view, in order to produce queries against the original RDF/S data that can actually be evaluated (thus avoiding the computation of the view data in its entirety). In relational databases, composing SQL queries with SQL view definitions is fairly straightforward. Composing RQL queries with RVL views is more challenging [18].

Consider, for example, the following query, which retrieves the database courses created by the author named "Professor A":

```
SELECT Y
FROM {X}myview:creates{Y}, {X}myview:name{Z}
WHERE Z = "Professor A"
USING NAMESPACE myview=&http://www.ics.forth.gr/LO.rdf#
```

which translates to:

```
ans(Y) :- {X}creates{Y}, {X}name{Z}, Z="Professor A"
```

This rule is translated into the RQL/RVL internal model *of the view*, as follows:

```
ans(y) :- P_SUB_V(q',creates),P_EXT_V(x,q',y),
          P_SUB_V(q",name),P_EXT_V(x,q",z), z="Professor A"
```

Having translated both the query and the view definition into the internal logical framework, we can then compose this query with the view specification (all expressed in the internal models) by just performing a composition of (nonrecursive) Datalog programs. Note that P_SUB_V($q', creates$) matches only the reflexivity instance P_SUB_V($creates, creates$) (similarly for P_SUB_V($q'', name$)). Thus, we obtain:

```
ans(y) :- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
          C_SUB(Course,a1),C_EXT(Course,y),
          P_SUB(q2,subject),P_EXT(y,q2,z1),z1="Database Management",
          PROP(c1,createdBy,d1),P_SUB(r1,createdBy),P_EXT(v,r1,x),
          C_SUB(Course,c1),C_EXT(Course,v),
          P_SUB(r2,subject),P_EXT(v,r2,u),u="Database Management",
          P_SUB(r3,name),P_EXT(x,r3,z),
          z="Professor A"
ans(y) :- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
          C_SUB(Course,a1),C_EXT(Course,y),
          P_SUB(q2,subject),P_EXT(y,q2,z1),z1="Database Management",
          PROP(c1,createdBy,d1),P_SUB(r1,createdBy),P_EXT(v,r1,x),
          C_SUB(Course,c1),C_EXT(Course,v),
          P_SUB(r2,subject),P_EXT(v,r2,u),u="Programming",
          P_SUB(r3,name),P_EXT(x,r3,z),
          z="Professor A"
ans(y) :- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
          C_SUB(Course,a1),C_EXT(Course,y),
          P_SUB(q2,subject),P_EXT(y,q2,z1),z1="Programming",
          PROP(c1,createdBy,d1),P_SUB(r1,createdBy),P_EXT(v,r1,x),
          C_SUB(Course,c1),C_EXT(Course,v),
          P_SUB(r2,subject),P_EXT(v,r2,u),u="Database Management",
          P_SUB(r3,name),P_EXT(x,r3,z),
          z="Professor A"
ans(y) :- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
          C_SUB(Course,a1),C_EXT(Course,y),
          P_SUB(q2,subject),P_EXT(y,q2,z1),z1="Programming",
          PROP(c1,createdBy,d1),P_SUB(r1,createdBy),P_EXT(v,r1,x),
          C_SUB(Course,c1),C_EXT(Course,v),
          P_SUB(r2,subject),P_EXT(v,r2,u),u="Programming",
          P_SUB(r3,name),P_EXT(x,r3,z),
          z="Professor A"
```

As we can observe, the answer to the query posed on the view is the union of four queries, which we call -in order of listing- Q_1, Q_2, Q_3 and Q_4. Then, Q_1 and Q_4 minimize to Q_1' and Q_4', respectively:

```
ans(y) :- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
          C_SUB(Course,a1),C_EXT(Course,y),
          P_SUB(q2,subject),P_EXT(y,q2,z1),z1="Database Management",
          P_SUB(r3,name),P_EXT(x,r3,z),
          z="Professor A"
ans(y) :- PROP(a1,createdBy,b1),P_SUB(q1,createdBy),P_EXT(y,q1,x),
          C_SUB(Course,a1),C_EXT(Course,y),
          P_SUB(q2,subject),P_EXT(y,q2,z1),z1="Programming",
          P_SUB(r3,name),P_EXT(x,r3,z),
          z="Professor A"
```

However, Q'_1 is a subquery of (hence it contains) Q_3. Similarly, Q'_4 contains Q_3. Hence, the entire program can be minimized to $Q'_1 \cup Q'_4$! The result of translating this reformulated query back to RQL would be:

```
ans(Y) :- {Y;Course}createdBy{X},
          {Y}subject{Z1}, Z1="Database Management",
          {X}name{Z}, Z="Professor A"
ans(y) :- {Y;Course}createdBy{X},
          {Y}subject{Z1}, Z1="Programming",
          {X}name{Z}, Z="Professor A"
```

Thus, the query posed on the view is translated into a query posed to the source RDF/S description base, which can be actually evaluated (thus avoiding the computation of the view data in its entirety).

4 RQL/RVL Query Processing in SeLeNe

The query service is one of the core and most important services in the SeLeNe system as described in Deliverable 3 [52]. The query service is responsible for processing RQL/RVL queries posed to the system over one or more autonomous RDF description bases.

An important factor that determines the implementation of query processing is related to the distribution of the data, which in the case of a SeLeNe are RDF descriptions and RDF schemas of LOs. There are three different architectural alternatives for SeLeNe, with different degrees of distribution of the data, as described in Deliverable 5 [53]. These should all be examined in order to discover implications for the implementation of the SeLeNe query system. These alternatives are briefly discussed below:

- **Centralized:** In this scenario there is a single location where the RDF descriptions and schemas are found in their entirety. This location can be a single machine or a computational cluster, which consists of physically adjacent machines. All client nodes are aware of and send query requests only to this location. An example of a centralized system is a web site that contains all the courses of a university or a certain department.
- **Mediated:** In this scenario the functionality, and consequently the query service, is primarily facilitated by mediator nodes that are affiliated with a number of provider nodes. Providers store their own autonomous description bases, whose schemas must register to a

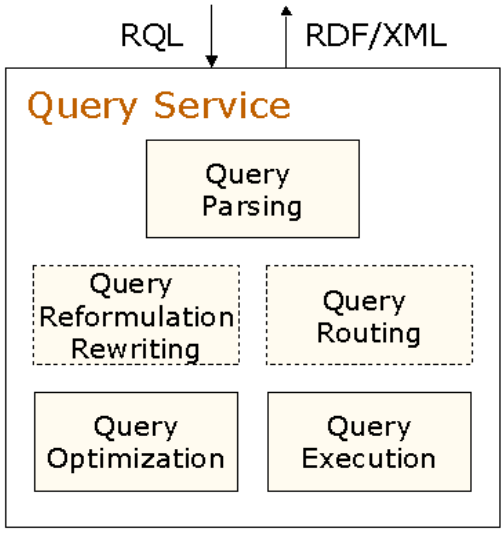

Figure 5: Query Service Architecture

mediator. These mediator nodes act as entry points to the SeLeNe system and offer directions as to where the required RDF descriptions and schemas can be found and accessed. Client nodes have to contact a known mediator, which in turn will contact the necessary providers and distribute the query requests accordingly. An example of a mediated architecture is a portal with educational material from various universities and other educational research organizations.

- **Autonomic:** In this scenario each peer node acts autonomously in providing RDF descriptions and query processing. RDF descriptions are maintained locally by each peer base and there is no global knowledge about the available peer bases. Since each node can enter and leave the system at will we need a flexible query routing and processing mechanism to satisfy client requests. An example of this type of architecture is a world wide self e-learning system where everyone is capable of publishing and retrieving educational material.

For each of the above scenarios there will need to be a different query service implementation, taking into account both the involved distribution and heterogeneity of the RDF resources and schemas.

In general, the query service can be seen as a set of modules that cooperate in order to support the full functionality of the service. It comprises (a) the parsing of the query, (b) the computation of possible routing of the query through the SeLeNe peer nodes, (c) the rewriting of the query in terms of a different descriptive schema, (d) the optimization of the query to reduce execution time and (e) finally the execution of the query in one or more description bases. The above functionalities can be seen as modules that together constitute the recommended query service architecture in Figure 5. The routing and the rewriting module are illustrated using dashed lines because they may not be necessary in all the architectural alternatives (i.e., for different architectural alternatives). For the rewriting module we also need appropriate mapping rules expressed in terms of RVL view definitions. These rules define how the rewriting of the

queries will be performed on RDF description bases and schemas. The routing module requires appropriate advertisements of description bases on remote peer nodes. In the following subsections we discuss how the query service is implemented in each of the architectural alternatives.

4.1 Centralized

In a centralized SeLeNe system the query processing mechanism is trivial, since a set of "fixed" peer nodes, i.e. the authority servers, handle the entire query processing load. Every query posed by a client is sent directly to the centralized RDF description base for evaluation. The authority servers have global knowledge of the system and can use it to create and utilize a global query plan that will return a correct and complete answer to the client request.

The query service in this scenario is comprised of the *parsing* module, for parsing the given query, the *optimization* module, for optimizing the query using appropriate indices, cost models, algebraic transformations, and the *execution* module, for obtaining the query results from the centralized base. These three modules can be described as the core modules, since they offer the basic functionality requested by a query service. The routing and the rewriting module are not used since the central base holds the totality of RDF descriptions that conform to one or more specific descriptive schemas (ontologies and taxonomies) with simple articulations (e.g., IsA relations).

RDFSuite [3] can support the above functionality and it is a good candidate for implementing the query service in this architectural alternative.

4.2 Mediated

In this scenario mediator nodes are primarily responsible for processing the queries since they are aware of the relevant provider nodes and how to access their RDF description bases. Each client node poses its queries to a known mediator, which is responsible for handling query requests in the most appropriate way. More precisely, a query expressed in terms of a mediator's schema needs to be reformulated in terms of the schemas employed by the local bases of the providers using the mapping rules. Thus, all the core modules of Figure 5, and the reformulation one, are needed for the mediated query service.

The Semantic Web Integration Middleware (SWIM) can provide the above functionality, since it supports mappings that can be utilized to support query mediation functionality required by this scenario. Readers are referred to [18] for more details on SWIM.

4.3 Autonomic

The autonomic scenario suggests a P2P-like architecture where each peer node disposes its own RDF description base without a central point of access as in the mediated scenario. Resource descriptions are accessible as long as the provider node is connected to the system. To implement query service in this scenario we need a distributed query processing mechanism [40]. This mechanism must be capable of migrating query plans from one peer node to another, and should adapt these plans to the characteristics of any particular SeLeNe (e.g., available peer nodes, bandwidth, etc.). In addition to the core modules for implementation of the query service, in a SeLeNe where only a partial knowledge about the RDF descriptions and schemas of peers is

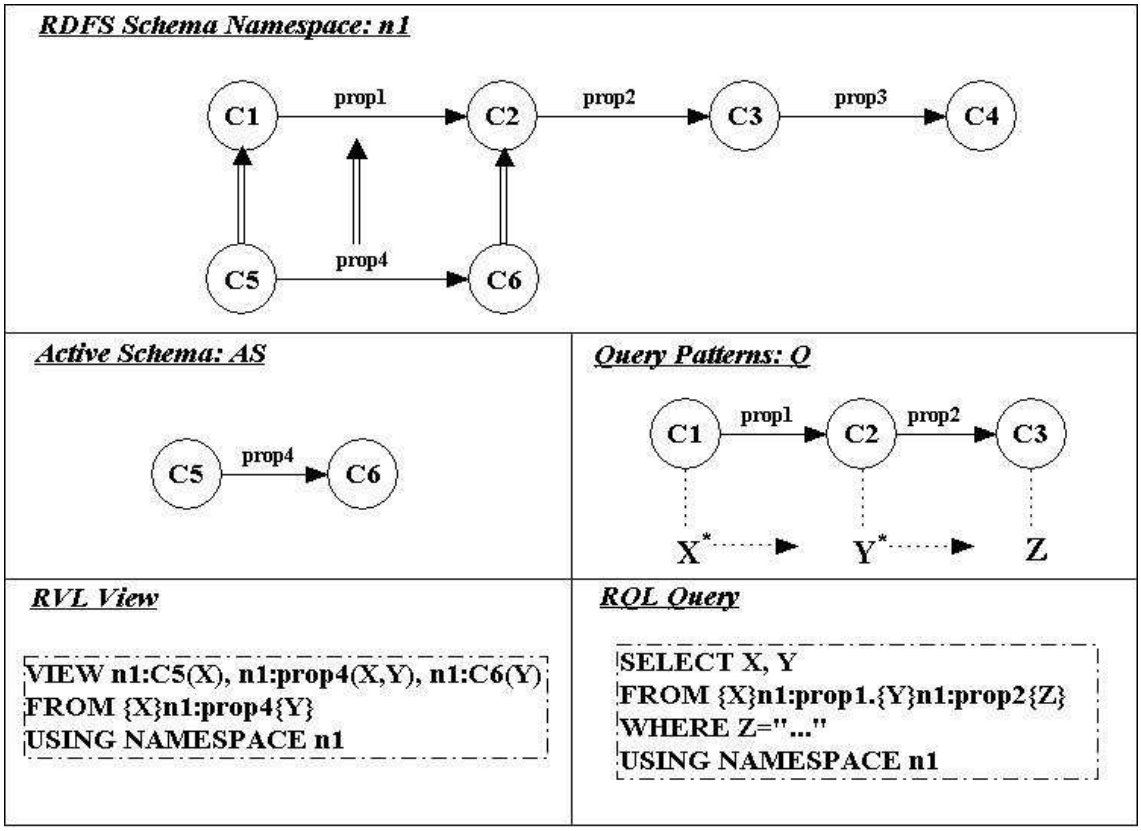

Figure 6: RDF/S Schema Namespace, Peer Active-Schema and Query Pattern Graphs

available, we will also need the routing module to dispatch query requests . The rewriting module could be also used for reformulating queries in terms of the local RDF schemas employed by peer nodes using appropriate mapping rules and views. We propose the following P2P semantic query routing and processing mechanism, based on the ICS-FORTH SQPeer Middleware, and we sketch the necessary constructs for its implementation.

In order to design an effective query routing and processing middleware for peer RDF/S bases, we need to address the following issues:

1. How peer nodes formulate queries?
2. How peer nodes advertise their bases?
3. How peer nodes route a query?
4. How peer nodes process a query?
5. How distributed query plans are optimized?

In the following subsections, we will present the main design choices for SQPeer in response to the above fundamental issues.

4.3.1 RQL Peer Queries

Each peer node in SQPeer provides RDF descriptions that conform to a number of RDF schemas. The notion of Semantic Overlay Networks (SONs) [21] appears to be an intuitive way to group

together peers sharing the same schema information. Peer nodes with the same schema can be considered to belong to the same SON. In a more sophisticated scenario, peer nodes contributing RDF descriptions specified by an RVL view are members of the same SON. This approach facilitates query routing, since a peer can easily identify relevant peers before sending query requests on the network. In the upper part of Figure 6 we can see an example of the schema graph of a specific namespace (i.e., n1) with four classes, `C1`, `C2`, `C3` and `C4`, that are connected with three properties, `prop1`, `prop2` and `prop3`. There are also two subclasses, `C5` and `C6`, of classes `C1` and `C2`, respectively, which are connected with sub-property `prop4` of property `prop1`.

Queries in SQPeer are formulated by client-peers in RQL, according to the RDF schemas they use to create their description bases or to define virtual views over their legacy databases. Thus, we need to capture intensionally the meaning of peer queries, in order to reason about query/view containment and guide query routing through the peer bases of the system. To this end, we introduce the notion of ***query patterns***.

An RQL query pattern graph, which describes the schema information employed by a query, is created by identifying the involved path patterns. In the rest of this deliverable we focus only on conjunctive RQL queries. In the bottom right corner of Figure 6 we can see an RQL query returning all the resources represented by the variables `X` and `Y`. In the `FROM` clause the employed path patterns imply a join on the `Y` resource variable between the target resource of the property `prop1` and the origin resource of the property `prop2`. The `WHERE` clause filters the returned resources according to the value of variable `Z`. Figure 6 illustrates the query pattern graph extracted by the above query, where `X` and `Y` resource variables are marked with "*" to denote projections. A graphical end-user interface may be used to create and visualize such query pattern graphs in a user-friendly way.

4.3.2 Peer Base Advertisement

In the context of a SON, each peer node should be able to advertise its description base in order to be discovered by other peers. Peer base advertisement in SQPeer relies on virtual or materialized RDF schema(s). Since these schemas contain numerous RDF classes and properties not necessarily populated with data in a peer base, we need a fine-grained notion of schema advertisements. The ***active-schema*** of a peer node is essentially a subset of the employed RDF schema(s) for which all RDF classes and properties are (in the materialized scenario) or can be (in the virtual view scenario) populated. The active-schema may be broadcast to (or requested by) other peer nodes, thus informing the rest of the P2P system of what is exactly available inside the peers' bases. A same type of peer advertisements is foreseen in Deliverable 4.4 [48], where an annotated RDF schema is kept in each peer for declaring what type of RDF data are stored in its local base.

The bottom left part of Figure 6 illustrates the RVL statement of an active-schema used to describe the data contained in a peer base. This statement "populates" the classes `C5` and `C6` and the property `prop4` (in the `VIEW` clause) with appropriate instances copied from the peer's local base (in the `FROM` clause). In the middle-left part of the figure we can see the corresponding active-schema graph obtained by this view. We note the similarity in the representation of active-schemas and query pattern graphs. This view can be a materialized RDF/S schema with actual resource descriptions or can be a virtual one, which can be populated with data from a relational or an XML peer base.

A more complex example is illustrated in Figure 7, comprising four peers and their corre-

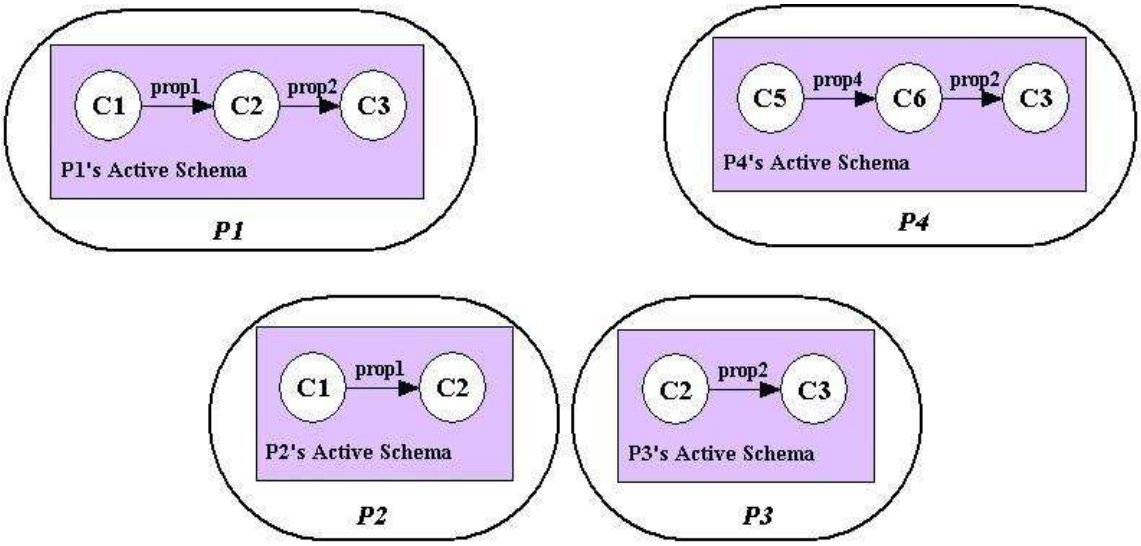

Figure 7: Example of Peers and their Active-Schemas

sponding active-schemas. Peer P1 contains resources related through the properties `prop1` and `prop2`, while peer P4 contains resources related through the properties `prop4` and `prop2`. Peer P2 contains resources related by `prop1`, while peer P3 contains resources related by `prop2`.

Representing active-schemas and query pattern graphs in an intensional way makes it easier to maintain a distributed knowledge of the P2P system, while yielding significant performance gains. First, by representing in the same way what is queried by a peer and what is contained in a peer base, we can reuse the RQL query/RVL view subsumption techniques, as proposed in SWIM [18]. Second, compared to global schema-based advertisements, we expect that the load of queries processed by each peer is smaller, since a peer receives only relevant to its content queries. This also affects the amount of network bandwidth consumed by the P2P system.

4.3.3 Semantic Query Routing

Query routing is responsible for finding the relevant to a query peer bases. In other words, query routing depends on the data distribution (vertical, horizontal and mixed) of peers bases committing to a SON RDF/S schema.

An example of a vertical data distribution is exhibited by peers P2 and P3 in Figure 7. Peer P2 contains resources related by `prop1`, while peer P3 contains resources related by `prop2`. In order to answer the RQL query of Figure 6 the results of the subqueries have to be "joined". An example of horizontal data distribution can be seen between peers P1 and P4 of Figure 7. Since `prop4` is sub-property of `prop1` and `prop2` is contained in both peers, a complete answer to a user query, implies to "union" the results of the subqueries sent to these two peers. Thus, vertical distribution ensures correctness of query results, while horizontal distribution favours completeness.

SQPeer relies on intensional techniques for matching a query against the active-schemas of the peers. In particular, the query/view subsumption techniques of [18], are employed to determine which part of a query can be answered by an active-schema and rewrite the query sent to a peer

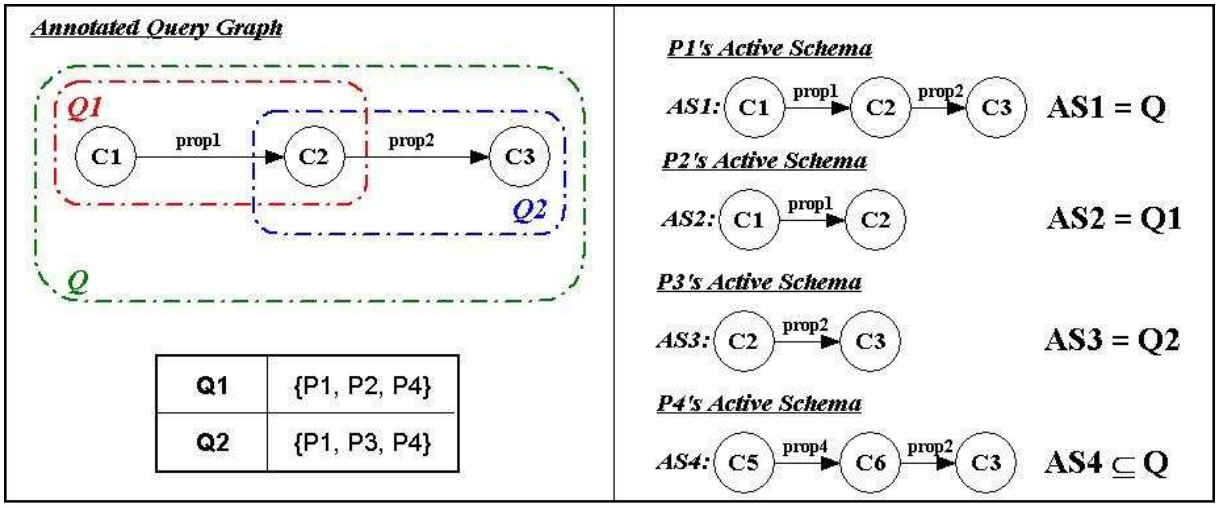

Figure 8: An Annotated RQL Query Graph

according to the available data in its base.

The query-routing algorithm takes as input a query graph and annotates each involved path pattern with the peers that can actually answer it, thus outputting an annotated query graph. A pseudocode description on how this algorithm works is given below.

Query-Routing Algorithm:

```
1. A peer P receives an RQL query Q.
2. Peer P parses the query Q and creates the corresponding query
   pattern graph by obtaining the involved paths.
3. For each pattern, the matching algorithm is performed.
   (a) Compare the path pattern with all known active-schemas.
   (b) If the active-schema graph is subsumed by the selected path
       pattern, then it is annotated with the name of the peer
       owning the active-schema.
4. Output annotated query graph.
```

An example of a query graph, which is composed of two path patterns, Q1 and Q2, is illustrated in Figure 8. In Figure 8 we can also see how the matching is performed with the active-schemas of the peers shown in Figure 7. P1's active-schema is equal to the path patterns Q1 and Q2, so both path patterns are annotated with P1. P2's active-schema is equal to path pattern Q1 and P3's active-schema is equal to Q2, so Q1 and Q2 are annotated with P2 and P3 respectively. Finally, P4's active-schema is subsumed by path patterns Q1 and Q2, since `prop4` is sub-property of `prop1`. Similarly to P1, Q1 and Q2 are annotated with P4. In the left part of Figure 8 we can see the annotated graph created by this matching.

4.3.4 Semantic Query Processing

Query processing in SQPeer takes the responsibility of generating distributed query plans according to the information returned by the query routing algorithm. These query plans are then

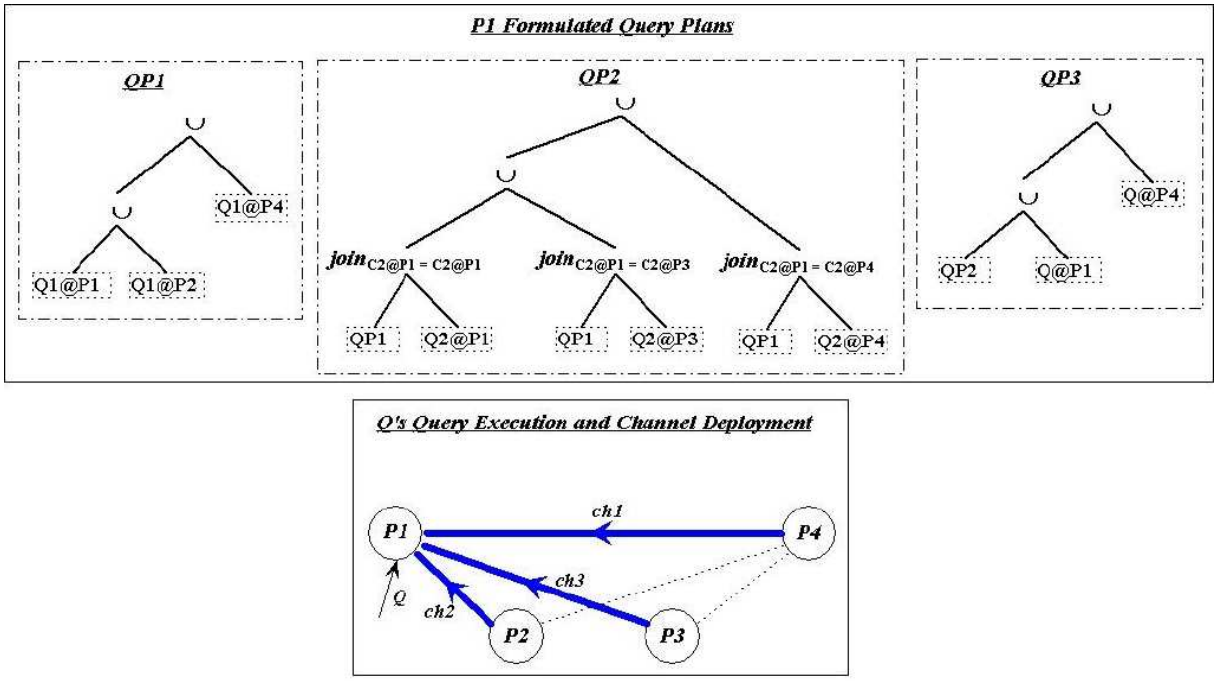

Figure 9: A SQPeer Query Execution Example

executed by the relevant peers. Communication channels enable this distributed execution and create the necessary foundation for exchanging appropriate data between the appropriate peers.

More precisely, ***channels*** [51] [58] are used in order to establish physical connections between remote peer nodes. Through channels, peers are able to route query plans and exchange the corresponding results according to the queries requested by client-peers. Adaptability during the execution phase (see Section 4.3.5) can be carried out by modifying the deployment of the channels through the system. Finally, channels permit each peer to further route and process the queries received, since it can be connected with more peers independently of the previous routing operations. Each channel has a root and a destination node. The root node of a channel is responsible for the management of the channel and for creating a locally unique id for it. Data packets are sent through each channel from the destination to the root node. Beside query results, these packets can also contain "changing plan" and failure information or other statistics useful for run-time query adaptability.

A query plan specifies how the query processing load should be distributed to the involved peers. Thus, the query plan highlights the way the channels are created in SQPeer. The query-processing algorithm receives as input an annotated query graph and returns as output its corresponding query plan. A pseudocode description on how this algorithm works is also given below.

Query-Processing Algorithm:

```
1. Peer P receives an annotated query graph AQ for the RQL query Q.
2. If P doesn't answer any part of the query Q
       Send query to a neighbour peer (see also Section 4.3.6);
```

```
   Else
      Create a new query plan QP;
3. Starting from a root of the graph, check every query path
   pattern PP in a sequential order, until AQ is fully processed;
4. If PP is the root
      Execute horizontal data distribution algorithm with input
      (QP, PP, AQ);
   Else
      Execute vertical data distribution algorithm with input
      (QP, PP, AQ);
      Execute horizontal data distribution algorithm with input
      (QP, PP, EQ ⋃ AQ), where EQ is the previously processed query;
5. Output formulated query plan QP.
```

Vertical data distribution algorithm with input (QP, PP, AQ):

```
1. Obtain set of peers, e.g., P'={P1,...,Pn}, from the annotated
   query graph that can answer PP.
2. For each peer of the set, e.g., Px, create query plan
   QPx = QP join_{Cp@P=Cq@Px} AQ@Px, where Cp and Cq are the classes
   on which the join of the two queries is executed.
3. Create query plan QP = QP1 ⋃ QP2 ⋃ ...  ⋃ QPn
```

Horizontal data distribution algorithm with input (QP, PP, AQ):

```
1. Obtain set of peers, e.g., P"={P1,...,Pn}, from the annotated
   query graph that can answer PP.
2. For each peer of the set, e.g., Px, expand query plan as QP = QP ⋃ AQ@Px
```

Figure 9 illustrates an example of how the RQL query Q, shown in Figure 6, can be executed over the P2P database system of Figure 7. The query is first sent to peer P1 which initially executes the query-routing algorithm in order to obtain the annotated query graph of Figure 8. P1 runs the query-processing algorithm and since it can answer a part of the query, creates a new query plan. The algorithm selects as a root of the annotated query graph, the path pattern Q1, for which it runs the horizontal distribution algorithm. This algorithm outputs query plan QP1, shown in Figure 9, since P1, P2 and P4 can execute query path pattern Q1. Next, path pattern Q2 is selected and the vertical data distribution algorithm executes and returns the query plan QP2. For each of the peers that can process Q2, we join the sub-queries sent with the results obtained from query plan QP1. P1 additionally follows the horizontal data distribution for obtaining more complete results. Since P1 and P4 can answer the whole query Q, P1's query plan will evolve to QP3: QP2 ⋃ Q@P1 ⋃ Q@P4. The final query plan and the deployment of the channels in SQPeer can also be seen in Figure 9.

Possible rewritings of the queries sent to remote peers in terms of different descriptive schemas may be necessary. This functionality can be implemented in the Semantic Web Integration Middleware (SWIM [18]), which supports powerful mappings to RDF/S of both structured relational and semistructured XML databases.

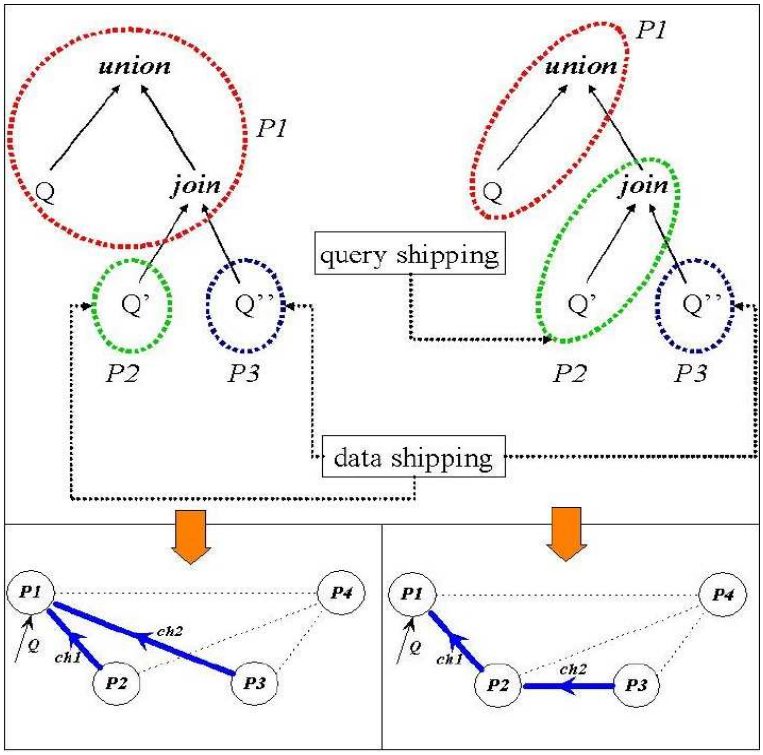

Figure 10: Data and Query Shipping Example

After the creation of the query plan, the peer holds responsibility for executing this plan and deploying the necessary channels in the system. From the query plan the peer obtains the set of peers that need to be contacted for executing the query. For each of these peers, a channel is created with the root being the peer executing the algorithm and the destination being the peer examined. Although each of these peers may contribute in the execution of the query plan by replying to more than one sub-query, only one channel is necessary. For the creation of the channels, the peer should also take into consideration certain optimization factors that are presented in the following subsection.

4.3.5 Query Optimization

In SQPeer we distinguish two possible optimizations of distributed query plans. First, compile-based optimization depends on statistics held by each peer and allows us to choose between different execution policies for the query plans (data or query shipping). These statistics involve response times from previously contacted peers or result sizes from previous executions of the same query. The type and the speed of the connection between the peers can be used to decide between different channel deployments. The processing load of the peers can be considered, since a peer that processes fewer queries, even if its connection is slow, may offer a better execution time. This processing load can be handled by the existence of slots in each peer, which show the amount of queries that can be handled simultaneously by this peer.

Once the query plan is generated, a peer node can decide at compile-time between data, query

or hybrid shipping execution policies. In the example of Figure 10 we can see two alternatives on how P1 handles the generated query plan. In the left part of the figure we can see the data shipping alternative, since P1 sends queries Q2 and Q3 to peers P2 and P3 and joins their results locally. In the right part of the query we can see the query shipping alternative, since P1 decides to forward the join operation down to P2, which in turn receives the results from P3 and executes the join locally before sending the full answer to P1 for further processing. At the bottom of the figure, we can see the deployment of the channels in SQPeer for each of these two alternative policies.

On the other hand, run-time adaptability of query plans is an essential characteristic of query processing when peer bases join and leave the system at will, or more generally when system resources are exhausted. For example, the optimizer may alter a running query plan by observing the throughput of a certain channel. This throughput can be measured by the number of incoming or outgoing tuples. Changing query plans may alter an already installed channel, as well as the query plans of the root and destination node of the channel. The root node of each channel is responsible for identifying possible problems caused by environmental changes and for handling them accordingly. It should also inform all the involved nodes that are affected by the alteration of the plan. Finally, the root node should create a new query plan by re-executing the routing and processing algorithms and not taking into consideration those peers that became obsolete.

We should keep in mind that switching to a different query plan in the middle of the query execution may cause some problems. Previous results, which were already created by the execution of the query at possible multiple peer nodes, have to be handled, since the new query plan will produce new results. Two are the possible solutions to this issue. The ubQL approach [51] proposes to discard previous intermediate results and all on-going computations are terminated. Alternatively [29] proposes a phased query execution, in which each time the query plan is changed the system enters into a new phase. The final phase, which is called the cleanup phase, is responsible for combining the sub-results from the other phases in order to obtain a full answer. In SQPeer middleware, we have adopted the ubQL approach.

4.3.6 SQPeer Architectural Alternatives

SQPeer can be used in different P2P architectural settings. Even though the P2P architecture affects the peers behaviour, our proposed query processing and routing algorithms work independently of the particular architectural setting.

On the one hand, we have ***client-peers***, which may frequently join or leave the system. These peers have only the ability to pose RQL queries to the rest of the P2P system. Since these peers usually have limited capabilities and they are connected for short periods of time, they do not participate in query routing and processing in the way the other peers do.

On the other hand, we may have ***simple-peers*** that act autonomously and may also join or leave the system but not so frequently as client-peers. Their corresponding RDF/S bases are available during their connection and may share their data with the rest of the system. Client-peers can contact them for posing queries that need to be processed in a distributed way. Along with their insertion in the P2P system, simple-peers should broadcast their active-schema information. Thus, a simple-peer identifies and connects physically with the SON(s) it belongs to and becomes known to its new neighborhood. Simple-peers have also the ability to pose queries to the system (as client-peers), but with the added functionality of executing these queries on

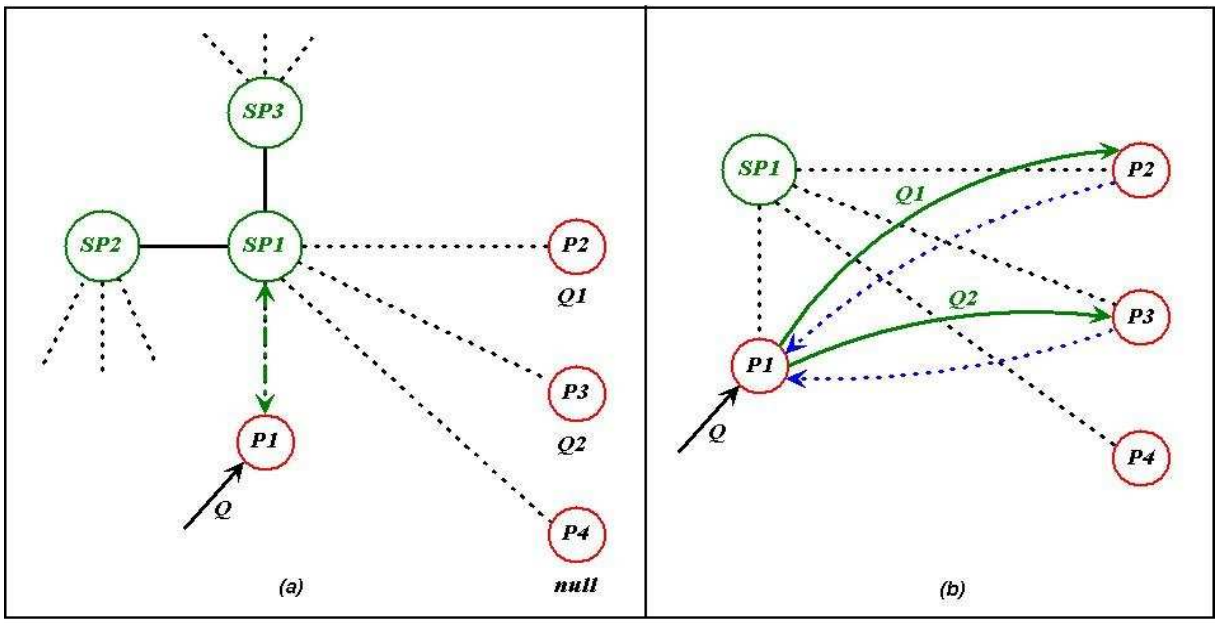

Figure 11: SQPeer Query Processing in a Super Peer-Like System

their own local base.

Finally, a small percentage of the peers may play the role of ***super-peers***. A super-peer acts as a centralized server for a subset of simple-peers. Super-peers are mainly responsible for routing queries through the system and for managing their group of simple-peers. Queries received and processed by a simple-peer are first sent to its super-peer, which handles the routing process and then sends the annotated query pattern back to the simple-peer for further processing and execution. Super-peers should be highly-available nodes and should offer high computing capabilities.

In this context we consider two architectural alternatives, distinguished according to the distribution of knowledge on a P2P system regarding peer base advertisements. The first corresponds to a hybrid P2P architecture based on the notion of Super-Peer Nodes (like Morpheus or Kazaa). The second is closer to an ad-hoc P2P architecture (like Freenet or Gnutella).

In a Super Peer-like system [60] [44] each peer is connected with at least one super-peer, who is responsible for collecting the active-schema information (materialized or virtual) of all its simple-peers. The peers that provide RDF descriptions for the same RDF/S schema are grouped under the same super-peer. Thus, each peer implicitly knows all the active-schemas of the peers that are semantically relevant to it in the sense that they employ the same RDF/S schema or view namespace. In this scenario, we can identify as neighbours of a peer the nodes belonging to the same super-peer, thus forming semantically related neighbourhoods. Each peer can be connected to multiple super-peers, since it can provide resource descriptions conforming to different namespaces. When a peer connects with a super-peer, it sends its corresponding active-schema (push). All super-peers are aware of each other in order to be able to answer queries expressed in terms of different RDF schemas. A client-peer can connect with a simple-peer and send it a query for further processing in the system. The simple-peer forwards the query to

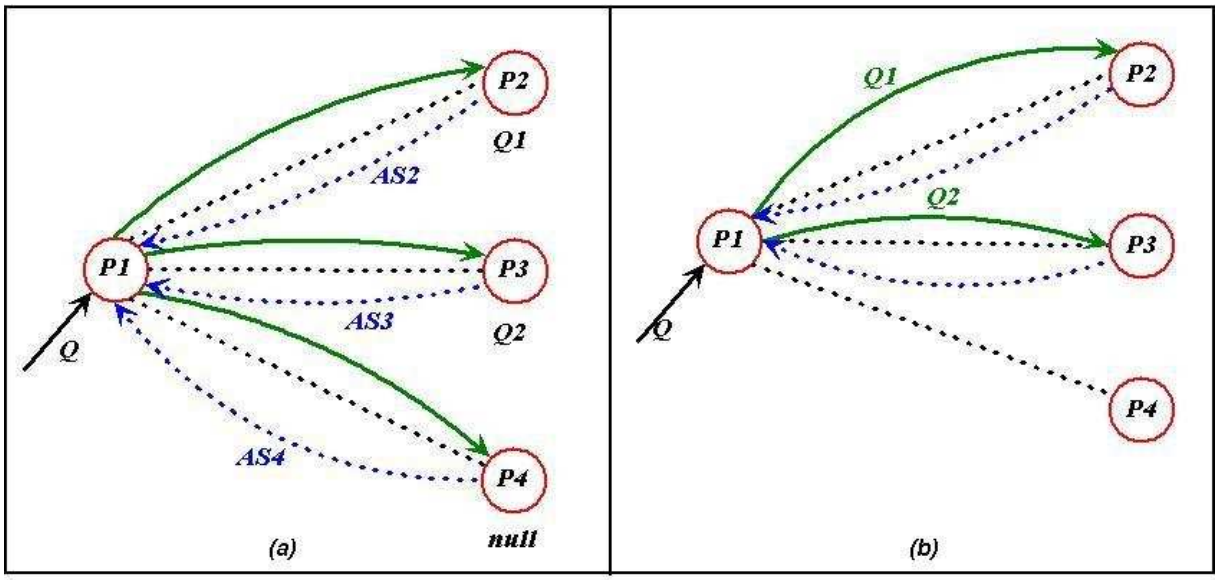

Figure 12: SQPeer Query Processing Mechanism in an Ad-Hoc P2P System

the super-peer according to the schema concerning the query. If this schema is unknown to the simple-peer, it sends the query randomly to one of its known super-peers, which then discovers the appropriate super-peer through the super-peers' backbone.

In Figure 11, we can see an example of how this scenario works. We consider a super-peer SP1 and a set of client peers, P1 to P4, which are connected with SP1. SP1 is also connected with other super-peers, for example SP2 and SP3, thus creating the necessary super-peer backbone. When P1 receives the query Q, it initially needs to send Q to SP1 (Figure 11a). SP1 searches the active-schemas of all its simple-peers and creates an annotated query graph containing the information that P2 can answer only the Q1 path pattern, while P3 can answer the Q2 path pattern. SP1 sends this annotated graph to P1 in order to generate the appropriate query plan and create the two channels with P2 and P3 for gathering the results (Figure 11b). P2 and P3 send their results back to P1, who joins them locally in order to produce the final answer. In this scenario, the query-routing algorithm is performed only by super-peers and the query processing and execution is left to the simple-peers.

Alternatively, we can consider an ad-hoc P2P architecture. In this alternative, when a peer first joins the system, it becomes aware only of its physically close neighbours and it can explicitly request their active-schemas that are related to its own RDF/S namespaces (pull). By requesting this information, the peer finds the relevant neighbourhood to its base and connects with it, thus creating self-adaptive SONs. Then, when a peer receives a relevant query, it can locally apply the query routing algorithm and create a query plan. If the peer receives a query, whose schema is unknown, it should broadcast the query to a neighbour peer, until a relevant one is found.

In Figure 12 we can see an example, where peers P1 to P4 are aware only of their own active-schemas. When P1 receives the query Q, since it does not know how to distribute it, it requests its neighbours' active-schemas, in order to apply the routing algorithm locally (Figure 12a). Since P2 can answer the Q1 part of Q and P3 can answer the Q2 part of Q, P1 creates two channels with those peers and sends them the corresponding queries. The results from those queries are

returned to P1 and are joined locally to produce a correct answer (Figure 12b).

These two architectural alternatives exhibit different behaviour in the routing, processing and execution of a query. In the ad-hoc architecture, SONs are created in a self-adaptive way, while in the super-peer architecture SONs are created in a more static way, since each super-peer is responsible for the creation and further management of SONs. The existence of SONs leads to minimizing the broadcasting (flooding) in the P2P system, since a query is received and processed only by the relevant peers in both architectures. It should be stressed that while in the ad-hoc architecture, peers handle both the query routing and processing load, super-peers are only responsible for routing and simple-peers for processing of the queries. Additionally, super-peers contain a global knowledge of the active-schemas of the peers in a SON and therefore can create a query plan offering completeness in the results. In the ad-hoc alternative, peers are aware only of a small number of active-schemas in the SON, and thus they can't guarantee result completeness. Finally, super-peers may handle the role of a mediator, as in the mediated scenario presented in Subsection 4.2.

5 Related Work

5.1 RVL and View Specification Languages

Several view specification languages have been proposed in the database literature. The most relevant to RVL is work conducted in the context of ODMG-compliant object-oriented DBMS, such as O_2 [2, 54], MultiView [50], Chimera [28] and K2 [57]. These view specification languages extend the relational approach for defining views as "named queries" with features for creating virtual object schemas. Apart from the differences between the ODMG and RDF/S data models (e.g., sub-properties, multiple classification of objects, etc.) and between the underlying design choices (e.g., in transformation expressiveness), the main novelty of RVL compared to these languages lies in its flexibility to create virtual classes (or properties) using RQL queries. This functionality is particularly useful for Semantic Web applications managing large schemas in a P2P way.

Some view specification languages have also been proposed for the RDF/S data model. In [59], set-based operations have been introduced in order to define object-preserving views using an untyped version of RQL. Unlike in RVL, the logical data independence of views is violated by this language, since virtual and source classes are merged into one global schema, while restructuring constructs for subsumption hierarchies are not supported. An alternative approach has been proposed in [23], which relies on F-logic rules to define only virtual description bases. Unlike RVL, this language does not provide the means to define virtual RDF/S schema graphs using, for instance, meta-schema instantiation capabilities. In the same spirit, [5] proposes a variation of RQL in order to produce as a query result an output RDF resource description graph instead of variable bindings in some tabular form. To the best of our knowledge, RVL is the first language offering a fully-fledged view specification for the RDF/S model.

5.2 RQL/RVL Query Processing and Routing

Several projects address query processing issues in P2P database systems. Query Flow [38] is a system offering dynamic and distributed query processing using the notion of HyperQueries.

HyperQueries are essentially sub-plans that exist in peer nodes and guide the processing of a query through the network. Furthermore, ubQL [51] provides a suite of process manipulation primitives that can be added on top of any traditional query language to support distributed query optimization. ubQL distinguishes the deployment from the execution phase of a query and supports adaptability of query plans during the execution phase. Both approaches require an a priori knowledge of the relevant to a query peers. However, this knowledge is not available in P2P database systems.

Mutant Query Plans (MQPs) [47] are logical query plans, where leaf nodes may consist of URN/URL references, or of materialized XML data. The references to resource locations (URLs) refer to peers where the actual data reside, while the abstract resource names (URNs) can be seen as the thematic topics of the requested data in a SON. MQPs are themselves serialized as XML elements and are exchanged among the peers. When a peer N receives a MQP M, N can resolve URN references, materialize URL references, evaluate or re-optimize MQP sub-plans, or just route M to another peer. When a MQP is fully evaluated, i.e., is reduced to XML code only, the result is returned to the *target* peer, which has initiated the query. The efficient routing of MQPs is preserved by information derived from multi-hierarchic topic namespaces, e.g., for educational material on computer science and for geographical information. Unlike SQPeer, this approach reduces the optimization opportunities of MQP by simply migrating possibly big XML fragments of partially evaluated query plans. In addition, it is not clear how sub-topics are employed for query routing.

AmbientDB [13] addresses P2P data management issues in a digital environment, i.e., audio players exchanging music collections. AmbientDB assumes the existence of a common global schema. However, each client-peer may contain its own schema as long as it provides the necessary mappings to the global one. The query processing mechanism is based on a three-level translation of a global query algebra into multi-wave stream processing plans, distributed over an ad-hoc and self-organizing P2P network. AmbientDB relies on the standard relational data model and algebra. Initially, a user query is posed in the "abstract global algebra" providing the standard relational operators for selection, join, aggregation and sort over abstract table types, i.e., the relational tables that need to be contacted. Then, this abstract query plan becomes concrete by instantiating the abstract table types with concrete ones, i.e., the local or distributed tables that exist in the peer bases. Finally, at the execution level, the concrete query plan is executed by selecting between different query execution strategies. AmbientDB P2P protocol is responsible for the query routing and relies on temporary (logical) routing trees, which are created on-the-fly and are subgraphs of the Chord network. Chord can also be used to implement clustered indices of distributed tables in AmbientDB as Distributed Hash Tables (DHTs). Each AmbientDB peer contains the index table partition that corresponds to it after hashing the key-values of all tuples in the distributed table. The user decides for the use of such DHTs, thus accelerating relevant lookup queries. The proposed routing protocol works in a global-schema environment, where the necessary mappings are handled by each peer. In a more complex scenario, where multiple and different schemas exist and peers pose queries in more than one schemas, a semantics-based routing functionality is not provided in advance.

Other projects address mainly query routing issues in SONs. In [25] indices are used to identify peers that can handle containment queries (e.g., in XML). For each keyword in the query, a peer node searches its indices and returns a set of nodes that can answer it. According to the operators used to connect these keywords, the peer node decides whether to union or intersect the sets of

relevant peers. In this approach, queries are directly sent to the set of peers returned by the routing algorithm with no further details on how a set of semantically related peers can actually execute a complex query involving vertical and horizontal distribution.

RDFPeers [17] is a scalable distributed RDF repository based on a P2P architecture. RDF-triple-storing nodes are used to store RDF triples at three peers in the network according to the subject, the predicate or the object value. An extension to Chord, called Multi-Attribute Addressable Network (MAAN), is used to store these triples and to route disjunctive, range and conjunctive multi-predicate queries to the appropriate peers. This approach ignores RDF schema information in query routing, thus not taking into consideration possible subsumption of classes or properties. In addition, distributed query processing and execution policies are not addressed.

The Edutella project [44] explores the design and implementation of a schema-based P2P infrastructure for the Semantic Web. In Edutella, peer content is described by different and extensible RDF schemas. Super-peers are responsible for message routing and integration/mediation of peer bases. The routing mechanism is based on appropriate indices to route a query initially within the super-peer backbone and then between super-peers and their respective peers. A query processing mechanism in such a schema-based P2P system is presented in [16]. Query evaluation plans (QEPs) containing selection predicates, compression functions, joins, etc., are pushed from clients to simple or super-peers where they are executed. Super-peers dispose an optimizer for generating partial query plans determining the parts of the query to be sent to the next (super-)peers and the operators to be locally executed for combining the results. The proposed query processing facility does not take into account the possible existence of subsumption relationships of RDF classes and properties. Additionally, this approach does not consider run-time adaptability of query plans.

Although the use of indices and super-peer topologies facilitate query routing, the cost of maintaining (XML or RDF) indices in each peer is important compared to the cost of maintaining active-schemas (i.e. views), as in the case of SQPeer.

6 Summary and Future Work

We have presented *RVL*, a language that brings a new kind of capability to the management of RDF/S metadata: users can create virtual schemas and resource descriptions customized to the needs of specific applications. By distinguishing the abstraction layers in an RDF/S application and by exploiting the RQL type system, RVL realizes virtual schema creation as the instantiation of appropriate metaclasses and achieves its target functionality through the use of only two operators: the *instantiation* and the *subsumption* operators.

Several issues need to be dealt with in order to fully support a view definition mechanism for RDF/S. An important issue is checking the *consistency* of view specifications, i.e., checking whether the graph they produce satisfies the constraints of our model. We wish to develop methods for consistency checking that avoid the naive approach, in which the entire view data is constructed and then validated. Furthermore, although we have argued for the benefits of defining virtual views, it is possible to implement an RVL engine that would actually compute and *materialize* the views. Such a capability would be of interest in metadata transformation applications where, for example, subsidiary but independently functioning portals are created from a given central one. This raises the classical problem of maintenance/update of materialized views, a complex problem long pondered upon by the database community. In the context of

RDF/S, this problem is even more interesting, due to the peculiarities of the data model.

Additionally, more issues remain open and require further investigation. Specifically, the cases dealt with in this deliverable are conjunctive RQL queries and conjunctive RVL view definitions. In both these cases, we obtain a translation into non-recursive Datalog programs, to which we can apply well-known optimization techniques. We intend to study the conditions under which similar results can be obtained for a broader class of RQL queries and RVL view definitions. Another issue is the exploitation of existing knowledge about the source schemas and data in order to perform further optimizations during the reformulation process. The RQL/RVL internal model can also accommodate constraints, such as those expressible in OWL [22]. It will be interesting to study the optimization potential that stems from the use of such constraints (e.g., uniqueness or disjointness constraints) in query reformulation and minimization.

There is a need for sophisticated RQL/RVL query routing and processing middleware to implement the query service in SeLeNe. To address these issues in the most difficult architectural alternative, i.e., an autonomic scenario, we have presented the SQPeer middleware. We discussed how conjunctive RQL queries expressed against a SON RDF/S schema are represented as semantic query patterns. For peer base advertisement we employ active-schemas, which denote the parts of a SON RDF/S schema which is actually (or can be) populated in a peer base. We sketch a semantic query routing algorithm which relies on query/view subsumption techniques to produce semantic query patterns annotated with routing information. We also demonstrate how SQPeer query plans are created and executed taking into account the data distribution in peer bases. Finally, we have discussed several compile and run-time optimization opportunities for SQPeer query plans, as well as possible architectural alternatives.

Finally, several issues remain open with respect to the implementation of the ICS-FORTH SQPeer middleware. Further research should be done on the optimization of distributed and adaptive queries in SQPeer borrowing ideas from related works [7] [46] [30]. We plan to study the trade-off between result completeness and processing load using the concepts of Top N (or Bottom N) queries [40]. In the same direction, we can use constraints regarding the number of peer nodes that each query is broadcasted and further processed.

References

[1] Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)

[2] Abiteboul, S., Bonner, A.: Objects and Views. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Denver, Colorado (1991) 238–247

[3] Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., Tolle, K.: The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In: 2nd International Workshop on the Semantic Web (SemWeb'01), in conjunction with Tenth International World Wide Web Conference (WWW10), pp. 1-13, Hongkong (2001)

[4] Association for Computing Machinery. The ACM Computing Classification System (1998 version), http://www.acm.org/class/1998/

[5] Aidministrator Nederland bv: SeRQL user manual, Version:0.4 (2003), http://sesame.aidministrator.nl/publications/SeRQL

[6] ANSI/X3/SPARC Study Group on Database Management Systems. Interim Report. ACM SIGMOD Bulletin 7, N2 (1975)

[7] Avnur, R., Hellerstein, J.M.: Eddies:Continuously Adaptive Query Processing. ACM SIGMOD, p.261-272, Dallas, TX (May, 2000)

[8] Berners-Lee, T., Fielding, R., Masinter, L.: Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396, http://www.ietf.org/rfc/rfc2396.txt

[9] Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. In: Scientific American (May, 2001), http://www.sciam.com/2001/0501issue/0501berners-lee.html

[10] Bernstein, P., Levy, A., Pottinger, R.: A Vision for Management of Complex Models. Microsoft Research Technical Report MSR-TR-2000-53 (2000), http://research.microsoft.com/scripts/pubs/view.asp?TR_ID=MSR-TR-2000-53

[11] Bloom, B.S., Krathwohl, D.R. (editors): Taxonomy of Educational Objectives – The Classification of Educational Goals: Handbook I, Cognitive Domain. Longman, New York (1956)

[12] Bray, T., Hollander, D., Layman, A.: Namespaces in XML. W3C Recommendation (1999)

[13] Boncz, P., Treijtel, C.: AmbientDB: Relational Query Processing in a P2P Network. In: Proceedings of the International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P), LNCS 2788, Springer Verlag (2003)

[14] Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E.: Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation (2000)

[15] Brickley, D., Guha, R.V.: Resource Description Framework Schema (RDF/S) Specification 1.0. W3C Candidate Recommendation (2000)

[16] Brunkhorst, I., Dhraief, H., Kemper, A., Nejdl, W., Wiesner, C.: Distributed Queries and Query Optimization in Schema-Based P2P-Systems. In: Proceedings of the International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P), Berlin, Germany (2003)

[17] Cai, M., Frank, M.: RDFPeers: A Scalable Distributed RDF Repository Based on a Structured Peer-to-Peer Network. In 13th International World Wide Web Conference (WWW), New York (2004)

[18] Christophides, V., Karvounarakis, G., Koffina, I., Kokkinidis, G., Magkanaraki, A., Plexousakis, D., Serfiotis, G., Tannen, V.,: The ICS-FORTH SWIM – A Powerful Semantic Web Integration Middleware. In: Proceedings of the First International Workshop on Semantic Web and Databases, Co-located with VLDB 2003, Humboldt-Universitat, Berlin, Germany (2003) 381–394

[19] Christophides, V., Plexousakis, D., Scholl, M., Tourtounis, S.: On Labeling Schemes for the Semantic Web. In: Proceedings of the 12th International World Wide Web Conference (WWW'03), Budapest, Hungary (2003)

[20] Clarke, I., Sandberg, O., Wiley, B., Hong, T.W.: Freenet: A Distributed Anonymous Information Storage and Retrieval System. In: Proceedings International Workshop on Design Issues in Anonymity and Unobservability, Volume 2009 of LNCS, pages 46–66. Springer-Verlag (2001)

[21] Crespo, A., Garcia-Molina H.: Semantic Overlay Networks for P2P Systems. Stanford Technical Report (2003)

[22] Dean, M., Connolly, D., Van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D., Patel-Schneider, P., Stein, L.A.: OWL Web Ontology Language 1.0 Reference. W3C Working Draft (2002)

[23] Decker, S., Sintek, M., Nejdl, W.: TRIPLE: A Logic for Reasoning with Parameterized Views over Semi-Structured Data. Technical Report (2002), http://www.kbs.uni-hannover.de/Arbeiten/Publikationen/2002/triple_views.pdf

[24] Fensel, D., Ding, Y., Schulten, E., Omelayenko, B., Botquin, G., Brown, M., Flett, A.: Product Data Integration in B2B E-Commerce. IEEE Intelligent Systems, **16**(4) (2001) 54–59

[25] Galanis, L., Wang, Y., Jeffery, S.R., DeWitt, D.J.: Processing Queries in a Large P2P System. In Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE) (2003)

[26] The Gnutella Protocol Specification, Version 0.4. Available at: http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf

[27] The Gnutella File-Sharing Protocol. Available at: http://gnutella.wego.com

[28] Guerrini, G., Bertino, E., Catania, B., Garcia-Molina, J.: A Formal Model of Views for Object-Oriented Database Systems. Theory and Practice of Object Systems, **3**(3) (1997) 157–183

[29] Ives, Z. G.: Efficient Query Processing for Data Integration. PhD Thesis, University of Washington (2002)

[30] Ives, Z. G., Levy, A. Y., Weld, D. S., Florescu, D., Friedman, M.: Adaptive Query Processing for Internet Applications. IEEE Data Engineering Bulletin, **23**(2) (2000) 19-26

[31] IEEE, Draft Standard for Learning Object Metadata. Available at: http://ltsc.ieee.org/doc/wg12/LOM_1484_12_1_v1_Final_Draft.pdf (2002)

[32] IEEE, Draft Standard for Learning Object Metadata. Available at: http://ltsc.ieee.org/wg12/LOM_WD6.pdf (2001)

[33] Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: RQL: A Declarative Query Language for RDF. In: Proceedings of the Eleventh International World Wide Web Conference 2002, Honolulu, Hawaii, USA (2002) 592–603

[34] Karvounarakis, G., Magkanaraki, A., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M., Tolle, K.: RQL: A Functional Query Language for RDF. In: The Functional Approach to Data Management – Modelling, Analyzing and Integrating Heterogeneous Data, P.M.D.Gray, L.Kerschberg, P.J.H.King, A.Poulovassilis (eds.), LNCS Series, Springer-Verlag (2003) 435–465

[35] The Kazaa file-sharing system. Available at : http://www.kazaa.com

[36] Keenoy, K., Levene, M., Peterson, D.: Personalisation and Trails in Self e-Learning Networks. SeLeNe Project Deliverable 4.2 (2003), http://www.dcs.bbk.ac.uk/selene/reports/Del4.2-1.4.pdf

[37] Keenoy, K., Papamarkos, G., Poulovassilis, A., Levene, M., Peterson, D., Wood, P.T, Loizou, G.: Self e-Learning Networks - Functionality, User Requirements and Exploitation Scenarios. SeLeNe Project Deliverable 2.2 (2003), http://www.dcs.bbk.ac.uk/∼ap/projects/selene/

[38] Kemper, A., Wiesner, C.: HyperQueries: Dynamic Distributed Query Processing on the Internet. In Proceedings of the International Conference on Very Large Data Bases (VLDB), Rome, Italy (2001)

[39] Klein, M.: Combining and Relating Ontologies: An Analysis of Problems and Solutions. In: Proceedings of the IJCAI'01 Workshop on Ontologies and Information Sharing, Seattle, USA (2001)

[40] Kossmann, D.: The State of the Art in Distributed Query Processing. ACM Computing Surveys, **32**(4) (2000) 422–469

[41] Lassila, O., Swick, R.: Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation (1999)

[42] Magkanaraki, A., Karvounarakis, G., Tuan Anh, T., Christophides, V., Plexousakis, D.: Ontology Storage and Querying. Technical Report 308, ICS-FORTH, Heraklion, Crete, Greece (2002)

[43] The Morpheus File-Sharing System. Available at: http://www.musiccity.com

[44] Nejdl, W., Wolpers, M., Siberski, W., Schmitz, C., Schlosser, M., Brunkhorst, I., Loser, A.: Super-Peer-Based Routing and Clustering Strategies for RDF-Based P2P Networks. In: Proceedings of the 12th International World Wide Web Conference, Budapest, Hungary (2003)

[45] Nilsson, M.: IEEE Learning Object Metadata RDF Binding. Available at: http://kmr.nada.kth.se/el/ims/md-lomrdf (2002)

[46] The Object-Globe Project. Available at: http://www.db.fmi.uni-passau.de/projects/OG/

[47] Papadimos, V., Maier, D., Tufte, K.: Distributed Query Processing and Catalogs for P2P Systems. In: Proceedings of the 2003 CIDR Conference (2003)

[48] Papamarkos, G., Poulovassilis, A., Wood, P.T.: ECA Rule Languages for Active Self e-Learning Networks. SeLeNe Project Deliverable 4.4 (2003), http://www.dcs.bbk.ac.uk/selene/reports/Del4.4-1.0.pdf

[49] Rigaux, P., Spyratos, N.: SeLeNe Report: Metadata Management and Learning Object Composition in a Self e-Learning Network. http://www.dcs.bbk.ac.uk/selene/reports/seleneLRI3.pdf (2003)

[50] Rundensteiner, E.: MultiView: A Methodology for Supporting Multiple View Schemata in Object-Oriented Databases. In: Proceedings of the 18th International Conference on Very Large Data Bases, Vancouver, Canada (1992) 187–198

[51] Sahuguet, A.: ubQL: A Distributed Query Language to Program Distributed Query Systems. PhD Thesis, University of Pennsylvania (2002)

[52] Samaras, G., Karenos, K., Christodoulou, E.: A Grid Service Framework for Self e-Learning Networks. SeLeNe Project Deliverable 3 (2003), http://www.dcs.bbk.ac.uk/selene/reports/Del3.pdf

[53] SeLeNe Consortium: An Architectural Framework and Deployment Choices for SeLeNe. SeLeNe Project Deliverable 5 (2003)

[54] Souza dos Santos, C., Abiteboul, S., Delobel, C.: Virtual Schemas and Bases. In: M. Jarke, J. Bubenko and K. Jeffery (editors): Proceedings of the Fourth International Conference on Extending Database Technology, St John's College, Cambridge, UK. Lecture Notes in Computer Science No. 779 (1994) 81–94

[55] Stojanovic, L., Staab, S., Studer, R.: e-Learning based on the Semantic Web. In: Proceedings of the WebNet 2001 World Conference on the WWW and the Internet, Orlando, Florida, USA (2001)

[56] Stratakis, M., Christophides, V., Keenoy, K., Magkanaraki, A.: E-Learning Standards. SeLeNe Project Deliverable 2.1 (2003), http://www.dcs.bbk.ac.uk/∼ap/projects/selene/

[57] Tannen, V., Davidson, S.B, Harker, S.: The Information Integration System K2. In: Bioinformatics: Managing Scientific Data, T. Critchlow and Z. Lacroix eds., Elsevier (2003)

[58] Traversat, B., Abdelaziz, M., Doolin, D., Duigou, M., Hugly, J. C., Pouyoul, E.: Project JXTA-C: Enabling a Web of Things. In: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS), Big Island, Hawaii (2003)

[59] Volz, R., Oberle, D., Studer, R.: Views for Light-Weight Web Ontologies. In: Proceedings of the ACM Symposium on Applied Computing SAC 2003, Melbourne, Florida, USA (2003)

[60] Yang, B., Garcia-Molina, H.: Designing a Super-Peer Network. In: Proceedings 19th International Conference Data Engineering (ICDE), IEEE Computer Society Press, Los Alamitos, CA (2003)

Appendix: RVL Typing Rules

The type system foreseen by RQL [33] specifies a set of types, namely the `metaclass of classes` (MC) (τ_{M_c}), `metaclass of properties` (MP) (τ_{M_p}), `class` (τ_C), `property` ($\tau_P[\tau,\tau]$), `resource URIs` (τ_U), `literal` (τ_L) (XML Schema data types), `bag` ({.}), `sequence` ([.]) and `alternative` ((.)) types. The notation $\tau_P[\tau,\tau]$ for property types indicates the exact type of its domain (metaclass and class types) and range (metaclass, class and literal types) (first and second position in the sequence). For brevity, we use the notation τ_P for property types. RVL extends this type system by specifying two more metaschema types, ω_C and ω_P, used by the instantiation operator to create user-defined metaclasses of classes and properties, respectively. The restrictions and inferences specified by RVL are captured by the typing rules presented in Table 1 (in the Appendix). Each rule represents the drawing of a conclusion (the part below the horizontal line) on the basis of a premise (the part above the horizontal line). For instance, rule `12` states that: "*If e is an expression of property type and* e_1 *and* e_2 *are expressions of types* τ_1 *(resource, class or property) and* τ_2 *(resource, class, property or literal) respectively, then* $e(e_1,e_2)$ *is a valid expression of type sequence of types* τ_1 *and* τ_2*. Otherwise, a type error is returned*".

Table 1: RVL Typing Rules

Operation	Typing Rule	
`MC creation`	$\frac{e_1 : \omega_C, e_2 : \tau, \tau \in \{string, \tau_{M_c}, \tau_{M_p}, \tau_C, \tau_P, \tau_U\}}{e_1(e_2) : \tau_{M_c}}$	(1)
`MP creation`	$\frac{e_1 : \omega_P, e_2 : \tau, \tau \in \{string, \tau_{M_c}, \tau_{M_p}, \tau_C, \tau_P, \tau_U\}}{e_1(e_2) : \tau_{M_p}}$	(2)
`Class creation`	$\frac{e_1 : \tau_{M_c}, e_2 : \tau, \tau \in \{string, \tau_{M_c}, \tau_{M_p}, \tau_C, \tau_P, \tau_U\}}{e_1(e_2) : \tau_C}$	(3)
`Property Creation`	$\frac{\begin{array}{c} e : \tau_{M_p}, e_1 : \tau_1, \tau_1 \in \{string, \tau_{M_c}, \tau_{M_p}, \tau_C, \tau_P\} \\ e_2 : \tau_2, \tau_2 \in \{\tau_{M_c}, \tau_{M_p}, \tau_C\}, e_3 : \tau_3, \tau_3 \in \{\tau_{M_c}, \tau_{M_p}, \tau_C, \tau_L\} \end{array}}{e(e_1, e_2, e_3) : \tau_P[\tau_2, \tau_3]}$	(4)
`MC subsumption`	$\frac{e_1 : \tau_{M_c}, e_2 : \tau_{M_c}}{e_1 < e_2 > : [\tau_{M_c}, \tau_{M_c}]}$	(5)
`MP subsumption`	$\frac{e_1 : \tau_{M_p}, e_2 : \tau_{M_p}}{e_1 < e_2 > : [\tau_{M_p}, \tau_{M_p}]}$	(6)
`Class subsumption`	$\frac{e_1 : \tau_C, e_2 : \tau_C}{e_1 < e_2 > : [\tau_C, \tau_C]}$	(7)
`Property Subsumption`	$\frac{e_1 : \tau_P, e_2 : \tau_P}{e_1 < e_2 > : [\tau_P, \tau_P]}$	(8)
`MC population`	$\frac{e_1 : \tau_{M_c}, e_2 : \tau_C}{e_1(e_2) : \tau_C}$	(9)
`MP population`	$\frac{e_1 : \tau_{M_p}, e_2 : \tau_P}{e_1(e_2) : \tau_P}$	(10)
`Class population`	$\frac{e_1 : \tau_C, e_2 : \tau_U}{e_1(e_2) : \tau_U}$	(11)
`Property population`	$\frac{e : \tau_P, e_1 : \tau_1, \tau_1 \in \{\tau_U, \tau_C, \tau_P\}, e_2 : \tau_2, \tau_2 \in \{\tau_U, \tau_C, \tau_P, \tau_L\}}{e(e_1, e_2) : [\tau_1, \tau_2]}$	(12)