RAL: an Algebra for Querying RDF

Flavius Frasincar (flaviusf@win.tue.nl), Geert-Jan Houben (houben@win.tue.nl), Richard Vdovjak (richardv@win.tue.nl) and Peter Barna (pbarna@win.tue.nl)
Eindhoven University of Technology, Department of Mathematics and Computer Science, PO Box 513, NL-5600 MB Eindhoven, the Netherlands

Abstract. To make the World Wide Web machine-understandable there is a strong demand both for languages describing metadata and for languages querying metadata. The Resource Description Framework (RDF), a language proposed by W3C, can be used for describing metadata about (Web) resources. RDF Schema (RDFS) extends RDF by providing means for creating application specific vocabularies (ontologies). While the two above languages are widely acknowledged as a standard means for describing Web metadata, a standardized language for querying RDF metadata is still an open issue. Research groups coming both from industry and academia are presently involved in proposing several RDF query languages. Due to the lack of an RDF algebra such query languages use APIs to describe their semantics and optimization issues are mostly neglected. This paper proposes RAL (an RDF algebra) as a reference mathematical study for RDF query languages and for performing RDF query optimization. We define the data model, we present the operators to manipulate the data, and we address the application of RAL for query optimization. RAL includes: extraction operators to retrieve the needed resources from the input RDF model, loop operators to support repetition, and construction operators to build the resulting RDF model.

Keywords: RDF(S), RDF(S) query language, RDF(S) algebra

1. Introduction

On the basis of its number of users and the attention it attracts, it is fair to say that the World Wide Web is the most popular source of information. With its overwhelming success and its considerable influence on the way in which we exchange information, some compare it to Gutenberg's invention of the printing press. Computer applications make information available for a very diverse audience on different platforms worldwide and 24 hours a day.

This first generation of the Web is strongly dependent on the human skills of the users. Computers support the users in providing them with the information on the screen or paper, but it is left to the users to read the information. In [7] Tim Berners-Lee describes a vision on a next-generation Web, called the Semantic Web (SW). In this vision he explains how the Web paradigm can be exploited in a context where computer applications deal with information. To make the information available for computer applications, first of all the content of the Web needs to be machine-readable. Moreover, in order to allow the computer applications to effectively process the content, it should be not only machine-readable but also machine-understandable. In the vision of the SW its final goal is to make machines fully exploit the Web knowledge in an uncentralized manner.

To reach this goal, the content (data) of the SW is accompanied by metadata. The metadata provides information about the content in order to support applications in their processing of the content, e.g. by capturing information about the meaning or semantics of the given data. W3C has a standard called Resource Description Framework (RDF) [26, 10] that is intended to serve as a metadata language for the Web and together with its extensions lays a foundation for the SW. The RDF language supports the expression of metadata. It has a graph notation, which can be serialized in a triple notation (subject, predicate, object) or in an XML syntax [2].

Compared to XML, which is document-oriented, RDF takes into consideration a knowledge-oriented approach that is designed specifically for the Web and that is extremely useful for the SW. One of the advantages of RDF over XML is that an RDF graph depicts in a unique form the information to be conveyed while there are several XML documents to represent the same semantic graph. The central concept that RDF uses in modeling the metadata is that of resource: resources act as the objects or entities that are considered in the metadata. RDF's purpose to express metadata is met by its ability to define statements that assign values to properties of resources. In this way RDF expressions describe how resources are related to each other and to (concrete) values.

Object-oriented systems are object-centric in the sense that properties are defined in a class context. On the contrary, RDF is property-centric, which makes it easy for anyone to "say anything about anything" [6], one of the architecture principles of the SW. In RDF, concepts from E-R modeling are being reused for the modeling of Web ontologies. The concept of ontology is used to express a common understanding of resources that allows application interoperability [14]: identifying a common structure of resources supports the uniform understanding and treatment of metadata.

The language of RDF is composed from different parts. RDF Schema (RDFS) [10] can be used to define application specific vocabularies. These vocabularies define taxonomies of resources and properties such that they subsequently can be used by specific RDF descriptions. RDFS is designed as a flexible language to support distributed description models. Unlike XML DTD or XML Schema [36, 8], RDFS does not impose a strict typing on descriptions: for example, one can use new properties that were not present in the schema, a resource can be an instance of more than one class, etc. The set of primitive data types in RDF is left on purpose poorly defined as it is envisaged that RDFS will reuse the work done for data typing in XML Schema [23]. We do hope that future versions of RDFS will bring clarification regarding RDF shortcomings of the present specification (e.g. missing set collection, difficult literals handling etc.).

In order to use metadata for application interoperability it is not sufficient to just have a language to describe the metadata. A language for describing queries on that data is also needed. In the XML world there is already a winner in the quest for the most appropriate XML query language, i.e. XQuery [9]. As the SW initiative started recently, its supporting technologies are still in their infancy. Research groups coming from both industry and academia are presently involved in proposing several RDF query languages (see the next section). We observe that such query languages often use APIs to describe their semantics. Clearly, for a proper understanding and a sound theoretical foundation of these query languages there is a lack of an algebra in the spirit of the one we know from the relational model. As we also observe that optimization issues are mostly neglected, an algebra for RDF could help to build a platform for finding efficient rewritings of queries. This paper identifies this need and proposes RAL, an RDF algebra suitable for defining (and comparing) the semantics of different RDF query languages and (at a later stage) for performing algebraic optimizations.

The remainder of this paper begins with discussing the related work on RDF query languages. In Section 3 the definition of RAL starts by considering its data model. Section 4 presents the definition of the basic operators of the algebra, while some additional algebra features are presented in the next section. Section 5 also shows how the algebra can be used to express queries from other query languages like RQL. Section 6 discusses RAL equivalence laws and their application for query optimization. Section 7 concludes the paper and indicates further research.

2. Related work

In the previous section we addressed the role of an algebra for the definition and comparison of query languages and for query optimization. At present, there already exist a few RDF query languages but to our knowledge there is no full-fledged RDF algebra. The only algebraic description of RDF that we encountered so far is the RDF data model specification [30] done by Sergey Melnik (Stanford). This specification is based on triples and it provides a formal definition of resources, literals, and statements. Despite being nicely defined, the specification does not include URIs, neglects the RDF graph structure, and does not provide operations for manipulating RDF models. Another formal approach, which aims not only at formalizing the RDF data model but also at associating a formal semantics to it, is the RDF Semantics (RS) [19]. However, it does not qualify as an algebraic approach but rather, as a model-theoretic one. As RS is currently being considered a main reference when it comes to RDF semantics, we tried to make our algebra (especially the data model part) compatible with RS.

As implementation of RDF toolkits started before having an RDF query language, there are a lot of RDF APIs present today. Three main approaches for querying RDF (meta)data have been proposed.

The first approach (supported in the W3C working group by Stanford) is to view RDF data as a knowledge base of triples. Triple [35], the successor of SiLRI (Simple Logic-based RDF Interpreter) [13], maps RDF metadata to a knowledge base in Horn Logic (replacing Frame Logic). A similar approach is taken in Metalog [28], which matches triples to predicates in Datalog, a subset of Horn Logic. In this way one can query RDF descriptions at a high level of abstraction: the querying takes place at a logical layer that supports inference [18].

The second approach (proposed by IBM) builds upon the XML serialization of RDF. In the "RDF for XML" project (recently removed), an RDF API is proposed on top of the IBM AlphaWork's XML 4 Java parser. In the context of the same project a declarative query language for RDF (RDF Query) [27] was created for which both input and output are resource containers. One of the nice features of this query language is that it proposes operators similar to the relational algebra, leaving the possibility to reuse some of the 25 years experience with relational databases. Unfortunately, the language fails to include the inference rules specific to RDF Schema, loosing description semantics.

Stefan Kokkelink goes even further with the second approach proposing RDF query and transformation languages that extend existing XML technologies. Similarly to XPath, he defines RDFPath [24] for locating information in an RDF graph. The location step and the filter constructs were present also in XPath, but the primary selection construct is new in this language. With the RDF graph being a forest, one needs to specify from which trees the selection will be made. RDFT is an RDF declarative transformation language a la XSLT [22], while RQuery, an RDF query language, is obtained by replacing XPath [5] with RDFPath in XQuery [9]. However, this approach is not using the features specific for RDF, as the RDF Schema is being completely neglected.

The third approach (coming from ICS-FORTH in Greece) uses the RDF Graph Model for defining the RDF query language RQL [21]. It extends previous work on semistructured query languages (e.g. path expressions, filtering capabilities etc.) [11] with RDF peculiarities. Its strength lies in the ability to uniformly query both RDF descriptions and schemas. Compared to the previous approach it exploits the inference given in the RDF Schema (e.g. multiple classification of resources, taxonomies of classes and properties etc.) making it the most advanced RDF query language proposed so far.

Other query languages for RDF have been proposed during the last years: we name Algae [33] (W3C) and rdfDB Query Language [17] (Netscape) as graph matching query languages. RDF query languages similar to rdfDB Query Language are: RDFQL [20], David Allsop's RDF query language [1], SquishQL [32], and RDQL [34] (HP Labs) an implementation of SquishQL on top of the Jena RDF API [29] of Brian McBride (HP Labs). Some other proposed RDF APIs are: Wilbur [25] (Nokia), the RDF API of Sergey Melnik [31] (Stanford), and Redland [3]. DAML Query Language (DQL), a query language for ontology knowledge expressed in DAML+OIL [12] (built on top of RDF), is currently under development.

We mention one characteristic aspect of all the languages. The proposed approaches disregard the (re)construction of the output: they leave the output as a "flat" RDF container of input resources. The focus is on the extraction of the proper resources for the given query, not on building a new RDF data structure. For the purpose of an RDF algebra we need to take into account also the construction part: deriving from the input data structure a new RDF data structure as the consequence of the query implies that the resulting RDF graph can contain new vertices and edges not present in the original RDF graph. To express RDF queries both the extraction and construction parts should be covered. This construction part is not only necessary for querying, but also for query optimization. The optimization of queries can be achieved not only in the extraction part, finding efficient ways of extracting the relevant resources, but also in the construction part when the actual output is produced.

3. Data model

In this section we discuss the data model used with our algebra. We describe how the RDF data structures are represented in the input or output of the expressions formulated in RAL. We start by considering the concept of RDF model.

3.1. RDF model

An RDF model is similar to a directed labeled graph (DLG) [26]. However, it differs from a classical DLG since its definition allows for multiple edges between two nodes. It also differs from a multigraph because the different edges between two nodes are not allowed to share the same label. The graph does not necessarily have to be connected and it is allowed to contain cycles.

The nodes in the graph are used to represent resources or literals. Literals (strings) are used to denote content that is not processed further by the RDF processor. The nodes that represent resources can be further classified as nodes representing URI references or blank nodes. URI references are used as universal identifiers in RDF. Each blank node, also called an anonymous resource, is considered to be unique in the graph despite the fact that it has no (explicit) label associated to it. The non-blank nodes are (explicitly) labeled with resource identifiers (URIs) or string values. The edges in the graph represent properties. These edges are labeled by property names. Edges between different pairs of nodes may share the same label and the same property can be applied repetitively on a certain resource. This RDF feature enables multiple classification of resources, multiple inheritance for classes, and multiple domains/ranges for properties. Both resources and properties are first class citizens in the proposed RDF data model.

We identify the following sets: R (set of resources), U (set of URI references), B (set of blank nodes), L (set of literals), and P (set of properties). At RDF level the following holds for these sets: $R = U \cup B$, $rdf{:}Property \in U$, $P \subset R$, $rdf{:}type \in P$, and U, B, and L are pair-wise disjoint.

The property *rdf:type* defines the type of a particular resource instance. At RDF level any resource can be the target of an *rdf:type* property. RDF supports multiple classification of resources, because *rdf:type* (as any other property) can be repeated on a particular resource.

Definition 1. An RDF model M is a finite set of triples (also called statements)

$$M \subset R \times U \times (R \cup L)$$

Each triple or statement in an RDF model contains a resource, a URI reference (which stands for a property), and a resource or literal.

Definition 2. The set of properties of an RDF model M is

$$P = \{\ p \mid (s, p, o) \in M \vee (p, rdf{:}type, rdf{:}Property) \in M\}$$

The properties in an RDF model are the middle element of a triple in the model, or they are a resource with an *rdf:type* property to the *rdf:Property* resource.

Definition 3. Formally the data model (graph model) corresponding to an RDF model M is

$$\begin{aligned} G &= (N, E, l_N, l_E) \\ l_N &= N \rightarrow R \cup L \\ l_E &= E \rightarrow P \end{aligned}$$

using the following construction mechanism (N and E denote the nodes and edges, l_N and l_E their labels). For each $(s, p, o) \in M$, add nodes n_s, n_o to N (different only if $s \neq o$) and label them as $l_N(n_s) = s$, $l_N(n_o) = o$, and add e_p to E as a directed edge between n_s and n_o and label that as $l_E(e_p) = p$. In the case that s and/or o are in B, then $l_N(n_s)$ and/or $l_N(n_o)$ are not defined: blank nodes do not have labels.

The function $l_N(.)$ is an injective partial function, while $l_E(.)$ is a (possibly non-injective) total function: nodes that have a label have a unique one, edges always have a label but can share it with other edges.

We use quotes for strings that represent literal nodes to make a syntactical distinction between them and URI nodes. A URI can be expressed by qualified names (e.g. *s:Painting*) or they can be in absolute form (e.g. *http://example.com/schema#Painting*). Blank nodes do not have a proper identifier which implies that they can be queried only through a property related to them. Compared to XML, which defines an order between subelements, in RDF the properties of a resource are unordered unless they represent items in a sequence container. We remark that not having the burden of preserving element order eases the definition of algebra operators and their associated laws.

3.2. Nodes and edges

As we describe in Table I each node has three basic properties. The *id* of a node represents the (identification) label associated to it. The nodes from the subset of resources that represent the blank nodes do not have an *id* associated to them. There are two *type*s of nodes: *rdfs:Resource* and *rdfs:Literal*. The *nodeID* gives the unique internal identifier of each node in the graph. *nodeID* has the same value as *id* for the nodes that have a label, but in addition it gives a unique identifier to the blank nodes. The internal identifier *nodeID* is not available for external use, i.e. it is not disclosed for querying.

Each edge has three basic properties as described in Table II. Compared with nodes, which have unique identifiers, edges have a *name*

Table I. Basic properties for nodes

Basic property	Result for resource $u \in U$	Result for literal $l \in L$
id	$l_N(u)$	$l_N(l)$
type	$rdfs{:}Resource$	$rdfs{:}Literal$
nodeID	internal ID	internal ID

(label), which may be not unique. There can be several edges sharing the same *name* but connecting different pairs of vertices. The *name* of an edge is (lexically) identified with the *id* of the resource corresponding to the property associated with the edge. The *subject* of an edge gives the resource node from which the edge is starting. *object* returns the resource or literal node where the edge ends, i.e. the value of the property.

Table II. Basic properties for edges

Basic property	Result for edge e from $r \in R$ to $o \in R \cup L$
name	$l_E(e)$
subject	r
object	o

Definition 4. Two non-blank nodes are considered to be equal if they have the same *id*. Two blank nodes are considered to be equal if they have the same (RDF) properties and the corresponding (RDF) property values are equal (in case of loops, pairs of blank nodes already visited are not further tested for equality).

All non-blank nodes that are considered equal are internally mapped into one node in the graph.

Definition 5. Two graphs are considered to be equal if they differ only by re-naming the *nodeID*s of their blank nodes.

Note that two graphs for which all their nodes are equal (in terms of node equality) may be not equal themselves (in terms of graph equality) if some corresponding non-blank nodes have different properties and/or different property values.

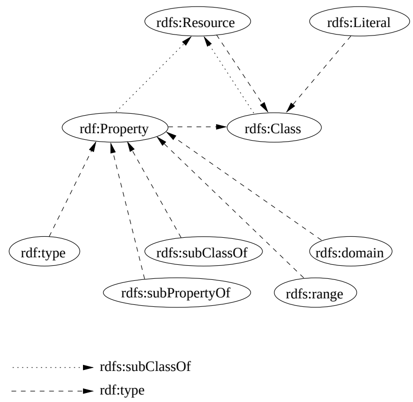

Figure 1. RDF/RDFS primitives.

3.3. RDFS

RDF Schema (RDFS) [10] provides a richer modeling language on top of RDF. RDFS adds new modeling primitives by introducing RDF resources that have additional semantics (in the previous section we already mentioned $rdfs{:}Resource$ and $rdfs{:}Literal$). If one chooses to discard this special semantics, RDFS models can be viewed as (plain) RDF models.

The RDFS type system is built using the primitives: $rdfs{:}Resource$, $rdf{:}type$, $rdf{:}Property$, $rdfs{:}Class$, $rdfs{:}Literal$, $rdfs{:}subClassOf$, $rdfs{:}subPropertyOf$, $rdfs{:}domain$, and $rdfs{:}range$. The distinction between rdf and $rdfs$ namespaces to be used for different resources is more due to historical reasons (RDF was developed before RDFS) than due to semantical ones. Figure 1 depicts graphically these RDF/RDFS primitives.

The inheritance mechanism incorporated in RDFS supports taxonomies at class level (using the property $rdfs{:}subClassOf$) and at property level (using the property $rdfs{:}subPropertyOf$). It also defines constraints: names to be used for properties, domain and range for properties etc. These constraints need to be fulfilled by RDF descriptions (later on called instances) in order to validate these instances according to the associated schema.

Every resource that has the $rdf{:}type$ property equal to $rdfs{:}Class$ represents a type (or class) in the RDF(S) type system. Types can be classified as primitive types ($rdfs{:}Resource$, $rdf{:}Property$, $rdfs{:}Class$, or $rdfs{:}Literal$) or as user-defined types (those are resources defined explicitly by a particular RDF model to have the $rdf{:}type$ property equal to $rdfs{:}Class$). The type of the resource $rdfs{:}Class$ is defined reflexively to be $rdfs{:}Class$. The resource $rdfs{:}Class$ contains all the types, which is not the same thing as saying that it includes all the values (instances) represented by these types.

We extend the data model with the set C (set of classes). At RDFS level the following holds: $C \subset R$, $rdfs{:}Resource \in C$, $rdf{:}Property \in C$, $rdfs{:}Class \in C$, and $rdfs{:}Literal \in C$.

Definition 6. The set of classes of an RDF model M is

$$C = \{\ c \mid (c, rdf{:}type, rdfs{:}Class) \in M\}$$

The most general types are $rdfs{:}Resource$ and $rdfs{:}Literal$ which represent all resources and literals, respectively. According to the data model these types are disjoint. Subclasses of the class $rdfs{:}Resource$ are $rdfs{:}Class$ and $rdfs{:}Property$, $rdfs{:}Class$ representing all types (already stated above) and $rdfs{:}Property$ containing all properties. The distinction between properties and resources is not a clear cut one as properties are resources with some additional (edge) semantics associated to them. A property (edge) can be used repetitively between nodes (similar in a way to repeating a particular type in the definition of its instances) which justifies the existence of an *extent* function (defined later on) for properties, as well as for classes. Moreover, property instances can have the $rdfs{:}subPropertyOf$ property defined in the same way as one can use the $rdfs{:}subClassOf$ property for classes.

The most important properties (each instance of $rdf{:}Property$) are: $rdfs{:}subClassOf$, $rdfs{:}subPropertyOf$, $rdfs{:}domain$, and $rdfs{:}range$. The properties $rdfs{:}subClassOf$ and $rdfs{:}subPropertyOf$ are used to define inheritance relationships between classes and properties respectively. Based on the RDF Test Cases [16] $rdf{:}subClassOf$ and $rdf{:}subPropertyOf$ can produce cycles, a useful mechanism if we think about class or property equivalence. A resource of type $rdf{:}Property$ may define the $rdfs{:}domain$ and the $rdfs{:}range$ associated to that property: the type of the subject and object nodes of the property edge. Inspired by ontology languages, like OWL [38], $rdfs{:}domain$ and $rdfs{:}range$ can be multiply defined for one particular property and will have conjunctive semantics.

There is one particular class called $rdfs{:}Literal$ that represents all strings. Note that the RDF Semantics [19] identifies two types of literals: plain literals and type literals. A plain literal is a 2-tuple (lexical form, language identifier) and a typed literal is a 3-tuple (lexical form, language identifier, datatype URI). The datatype URI is an XML Schema datatype [8] or $rdf{:}XMLLiteral$ for XML content. In the data model we simplify the literal definition considering just the character string (the lexical form) for literals. Note that literals are not resources, i.e. one cannot associate properties to them. On the other hand, there are resources that have type $rdfs$:*Literal* and thus can have properties attached to them. Nevertheless one cannot say which literal this resource denotes. RDF defines also the container classes rdf:*Seq*, rdf:*Bag*, and rdf:*Alt* to model ordered sequences, sets with duplicates, and value alternatives. The properties rdf:*rdf_1*, rdf:*rdf_2*, rdf:*rdf_3* etc. refer to container members.

3.4. Class and Property nodes

As shown in Table III each node representing a class has three schema properties. Schema properties associated to nodes are short notations (like a macro) for expressions doing the same computation based only on basic properties. The *type* of a class node is $rdfs$:*Class*. The set of superclasses (classes from which the current class node is inheriting properties) is given by *subClassOf*. RDFS allows multiple inheritance for classes because $rdfs$:*subClassOf* (as any other property) can be repeated on a particular class. The *extent* of a class node is the set of all instances of this class.

Table III. Schema properties for class nodes

Schema property	Result
type	$rdfs$:*Class*
subClassOf	S with $S \subset C$
extent	R' with $R' \subset R$

Each node representing a property has five schema properties as shown in Table IV. The *type* of a property node is rdf:*Property*. The set of superproperties (properties which the current property is specializing) is given by *subPropertyOf*. Note that the domain or range of a superproperty should be superclasses for the current property's domain or range, respectively. The *domain* and *range* return sets of classes that represent the domain and the range, respectively, of the property node. The *extent* of a property node is the set of resource pairs linked by the current property: this set of pairs is a subset of the Cartesian product between the associated domain and range extents.

One should note that we assume in the data model that there can be several edges having the same *name* but linking different pairs of resources. All these properties can be seen as “instances” (abusing the term “instance” previously referring to resource instances of a particu-

Table IV. Schema properties for property nodes

Schema property	Result
type	$rdf{:}Property$
subPropertyOf	S with $S \subset P$
domain	D with $D \subset C$
range	R with $R \subset C$
extent	E with $E \subset \cap_{d \in domain} extent(d) \times \cap_{r \in range} extent(r)$

lar class) of the property node with the *id* value equal to their common name.

In absence of a schema, all RDF properties have type $rdf{:}Property$, domain R, and range $R \cup L$. In this way one can define the *extent* of an RDF property even if the property is not explicitly defined in a schema. In a schemaless RDF graph all resources are assumed to be of type $rdfs{:}Resource$.

3.5. Complete models

The RDF Semantics [19] defines the RDF-closure and RDFS-closure of a certain model M by adding new triples to the model M according to a collection of given inference rules. We refer to the original model M as the extensional data and to the newly generated triples as the intensional data. There are two inference rules for RDF-closure and nine inference rules for RDFS-closure. The inference rules for RDF-closure add for all properties in the model the $rdf{:}type$ property (pointing to $rdf{:}Property$). Examples of inference rules for RDFS-closure are the transitivity of $rdfs{:}subClassOf$, the transitivity of $rdfs{:}subPropertyOf$, and the $rdf{:}type$ inference for an $rdf{:}type$ edge that follows after an $rdfs{:}subClassOf$ edge. One should note that the resulting output of applying these inference rules may trigger other rules. Nevertheless the rules will terminate for any RDF input model M, as there is only a finite number of triples that can be formed with the finite vocabulary of M.

Definition 7. An RDF model M is complete if it contains both its RDF-closure and RDFS-closure.

In the proposed data model we consider complete models and we neglect reification and the properties $rdfs{:}seeAlso$, $rdfs{:}isDefinedBy$, $rdfs{:}comment$, and $rdfs{:}label$ without loosing generality.

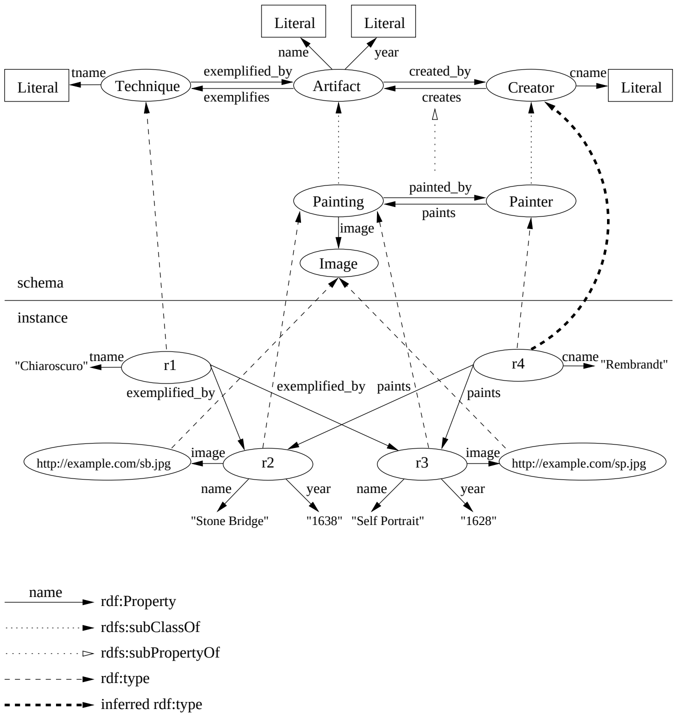

Figure 2. Example schema and instance.

4. Basic RAL operators

The purpose of defining RAL is twofold: to provide a reference mathematical study for RDF query languages and to enable algebraic manipulations for RDF query optimization. RAL is an algebra for RDF defined from a database perspective, some of its operators being inspired by their relational algebra counterparts. We used a similar approach in developing XAL [15], an algebra for XML query optimization.

During the presentation of RAL operators we will use the RDF data from the example in Figure 2 as input for the operators. It is assumed that all operators know about the complete RDF model as it was defined in Definition 7. That means that they all have the complete knowledge (both extensional and intensional data) present in the given model. Variants of the proposed operators can be defined using the suffix "ˆ" which will make the operators neglect the intensional data, i.e. data derived by applying RDF(S) inference rules to the input model is neglected (similar to RQL's "strict interpretation").

Figure 2 is an excerpt from the RDF schema and RDF instance of some Web data describing different painting techniques. For reasons of simplicity we consider only one painting technique ("*Chiaroscuro*"), one painter ("*Rembrandt*"), and two paintings of the same painter ("*StoneBridge*" and "*SelfPortrait*"). The figure does not present the RDFS primitives *rdfs*:*Resource*, *rdf*:*Property*, *rdfs*:*Class*, and *rdfs*:*Literal* from which all the resources and literals are derived. In order to simplify Figure 2 we chose to present only the extensional data and just one intensional data element given by the inferred edge *rdf*:*type* between $r4$ and $Creator$. For the same reasons we omit from the figure edges representing the inverse properties $exemplifies$ and *painted_by* between instances (e.g. the edge labeled *painted_by* between $r2$ and $r4$) that are nevertheless part of the data model.

We define RDF collections to be sets of nodes (resources/literals). A collection is denoted as $\{e_1, e_2, ...e_n\}$ where $e_1, e_2, ...e_n$ are the nodes in the collection. A node, a unique element in the RDF graph, is also a unique element in a collection that contains it. All RAL operators are closed for collections, which implies that RAL expressions can be easily composed. The collection concept is similar to the monad concept from mathematics [39]. A monad is defined over a certain type M. In contrast to the monad, RAL collections are more liberal in the sense that they are not restricted to a particular type M. A RAL collection can contain both literals and resources of different types. A monad is defined as a triple of functions $(map^M, unit^M, join^M)$. RAL also has the *map* operation defined and the monad $join$ operation is equivalent to RAL's *union* operation. In RAL there is no *unit* operation as the singleton collection $\{n\}$ is written in the same way as the single node n. Based on the similarities between monads and RAL collections, one can reuse the three monad laws (left unit law, right unit law, and associativity law) as equivalence rules in RAL (see the first three RAL laws from Section 6). The fact that RAL collections are not ordered enables the commutativity law of some binary operations (see Law 11 from Section 6). In comparison with the relational algebra, RAL is more powerful as binary operations like union do not have to meet the "compatibility" condition from the relational algebra.

RAL operators come in three flavors: extraction operators retrieve the needed resources from the input RDF model, loop operators support repetition, and construction operators build the resulting RDF model. The RAL philosophy is based on the fact that the collection of nodes represents a collection of graph components that contain these nodes. Using the extraction operators a subgraph of the original graph is selected. The construction operators build a new model by creating nodes/edges as well as reusing old nodes (possibly without some edges) and old edges.

The general form of the operators is

$$o[f](x_1, x_2, \dots x_n : expression)$$

Informally, this form represents the following. For each binding of x to a tuple from the input collections, $f(x)$ is computed. A tuple is formed by taking one element from each input collection: $x_1, x_2, \dots x_n$. Note that $x1, x2, \dots x_n$ are algebra expressions that return collections. f is a function that may use basic/derived properties or one of the proposed operators. Based on the semantics of operator o a partial result for the application of o to $f(x)$ is computed for each binding x. The operator result is obtained by combining (through set union) all partial results. All unary operators use this implicit union mechanism, the *map* operator, to compute the result. In the operator's general form, the function f is optional. For readability reasons we use for binary operators the infix notation.

RAL operators are defined to work on any RDF description, with or without an explicit schema. Note that implicitly there is always a default schema based on the following RDFS primitives: $rdfs$:*Resource*, rdf:*Property*, $rdfs$:*Class*, and $rdfs$:*Literal*. These RDFS primitives can be used to retrieve a particular schema in case that such information is not known in advance. Once the application schema is known, one can formulate queries to return instances from the input model.

4.1. Extraction operators

The extraction operators retrieve the resources/literals of interest from the input collection of nodes. If the operator is not defined on nodes that represent literals, these nodes are simply neglected.

In the examples that illustrate the operators we will use expressions that return collections of resources from the example RDF model m of Figure 2. The expression c represents the collection (set) of all resources present in model m.

Projection

$$\pi[re\_name](e : expression)$$

The input of the projection is a collection of nodes (specified by the expression e) and the projection operator computes the values (objects) of the properties with a name given by the regular expression *re_name* over strings. The symbol # represents the wildcard that matches any string.

Example 1. $\pi[exemplified\_by](r1)$ returns the collection of artifacts that exemplify the painting technique $r1$ from the input model (depicted in Figure 2): $r2$ and $r3$.

Example 2. $\pi[(P|p)aint[s]\#](r4)$ returns the collection of paintings painted by $r4$: $r2$ and $r3$.

Example 3. $\pi[rdf{:}type](r4)$ returns the collection of resources representing a type of $r4$: $Painter$, $Creator$, and $rdf{:}Resource$.

Selection

$$\sigma[condition](e : expression)$$

In a selection the *condition* is a Boolean function that uses as constants URIs and/or strings. The operators allowed in the *condition* are RAL operators, the usual comparison operators $(=, >=, <=, <, >, <>)$, and logical operators (and, or, not). The input of the selection is a collection of nodes and the operator selects only the nodes that fulfill the *condition*.

Example 4. $\sigma[\pi[tname] = \text{“}Chiaroscuro\text{”}](c)$ is a selection operation applied to the collection c of all resources in the input model. The expression returns the resource(s) representing the painting technique with the name “*Chiaroscuro*” (i.e. $r1$).

Example 5. $\sigma[\pi[rdf{:}type] = Creator](\{r3, r4\})$ returns resources from the input model with the value of $rdf{:}type$ being $Creator$: $r4$, since $r4$ is a resource of type $Painter$ and $Painter$ is a subclass of $Creator$.

Example 6. $\sigma\hat{}[\pi[rdf{:}type] = Creator](\{r3, r4\})$ (different from the selection in the previous example, as “ˆ” implies the use of only the extensional data) returns the empty collection, as the inferred $rdf{:}type$ of $r4$ (i.e. $Creator$) from the input model will not be available to the operator.

Cartesian product

$$(x : expression) \times (y : expression)$$

The Cartesian product takes as input two collections of nodes on which it performs the set-theoretical Cartesian product. Each pair of nodes is used to build an anonymous resource that has all the properties of the original resources. Thus, this newly built resource will have all the types of the original two resources (RDF multiple classification of resources). The final output is the collection of all those anonymous resources.

Example 7. $\sigma[\pi[rdf{:}type] = Technique](c) \times \sigma[\pi[rdf{:}type] = Painter](c)$ where c represents the collection of all resources in the input model, returns one anonymous resource having all the properties of the only technique $r1$ and the only painter $r4$. As a consequence this anonymous resource has both types $Technique$ and $Painter$.

Join

$$(x : expression) \ \bowtie [condition] \ (y : expression)$$

The join expression is defined to be a Cartesian product followed by a selection, so equivalent to

$$\sigma[condition](x \times y)$$

The expression has as input two collections of resources that have their elements paired only if they fulfill the *condition* (refering to the left and right operands). Anonymous resources are built for each such pair. The output is the collection of all those anonymous resources.

Example 8. $(t := \sigma[\pi[rdf{:}type] = Technique](c)) \bowtie [\pi[exemplified\_by](t) = \pi[paints](p)](p := \sigma[\pi[rdf{:}type] = Painter](c))$ where c represents the collection of all resources in the input model, returns an anonymous resource having all the properties of $r1$ and $r4$. Note that in this expression $r1$ and $r4$ are paired because there is a painting (e.g. $r2$) that exemplifies $r1$ and is painted by $r4$.

Union

$$(x : expression) \cup (y : expression)$$

The union operator combines two input collections of nodes reflecting the set-theoretical union.

Difference

$$(x : expression) - (y : expression)$$

The difference operator returns the nodes present in the first input collection but not in the second input collection.

Intersection

$$(x : expression) \cap (y : expression)$$

The intersection operator returns the nodes present in both input collections.

4.2. Loop operators

Loop operators are used in RAL to control the repetitive application of a function or operator. They express repetition at input and/or function/operator level.

Map

$$map[f](e : expression)$$

The map operator is defined as

$$\cup(f(e_1), f(e_2), ...f(e_n))$$

if the collection e contains the elements $e_1, e_2, ...e_n$. So, the map operator expresses repetition at input level. The results of applying the function/operator f to each element in the input collection are combined (through set union) to obtain the final result. All unary extraction operators have an implicit map operator associated with them.

Example 9. $map[id](c)$ where c represents the collection of all resources in the input model, computes the labels of all the non-blank nodes in the input model, i.e. the labels of all resources having an id property.

Kleene star

$$*[f](e : expression)$$

The Kleene star operator is defined as

$$e \cup f(e) \cup ...f(f(...(f(f(e)))...)) \cup ...$$

So, the Kleene star operator expresses repetition at function/operator level. It repeats the application of the function/operator f on the given input for possibly an infinite number of times. For each iteration the result is obtained by combining (through set union) the output of applying the function/operator on the input with the input. If after an iteration the result is the same as the input, a fixed point is reached and the repetition stops. In order to ensure termination, a variant of this operator that specifies the number of iterations n is defined below:

$$*[f, n](e : expression)$$

Note that the map operator does not include the input in the result, while the Kleene star operator does.

Example 10. $map[id](*[\pi[rdfs{:}subClassOf]](Painting))$ gives the id of all ancestor classes in the type hierarchy starting with $Painting$. For our example the result will contain three labels denoting the types $Painting$, $Artifact$, and $rdfs{:}Resource$. If there would have been loops made by the $rdfs{:}subClassOf$ property in the input model, the above example would still have terminated. The fact that the input model has a finite number of classes implies that at a certain moment a fixed point is reached (we obtain the same output collection as for the previous iteration) and thus the Kleene star operator terminates.

4.3. Construction operators

Querying an RDF model implies not only extracting interesting nodes from the input model but also constructing an output model by deleting nodes/edges from the extracted graph and by creating new nodes/edges.

Before actually committing a construction operation, the RDF constraints are checked on the output model. If these constraints are not met, the operation aborts. Examples of RDF constraints are: resource identifiers have to be unique, the value of *rdf:type* cannot be a literal, literals cannot have properties etc.

Create node

$$cnode[type, id]()$$

The create node operator possibly adds a new node to the graph. The input collection is not used in the operator semantics. The type of the new node, specified by *type*, is a resource of type $rdfs{:}Class$. The *id* is a resource identifier if the node represents a resource, or a string if the node represents a literal. The *id* is used as input in the system's new id generator (*nig*) skolem function. This function returns the unique $nodeID$. The $nodeID$ is equal to *id* if *id* is given or it is a new unique identifier if *id* is empty. In the first case an old node identifier is returned if *id* is already used as a $nodeID$ in the data model. In the second case a blank node is assigned a new $nodeID$. Note that the function *nig* is injective. As a side effect of this operator, an edge representing the *type* property is added between the newly created resource and its associated type resource. The create node operator returns the created node (a collection containing one node).

Example 11. $cnode[Painter]()$ creates a blank node of type Painter, while $cnode[Literal, \text{``}Caravagio\text{''}]()$ creates a Literal node representing the string "$Caravagio$".

Create edge

$$cedge[name, subject](object : expression)$$

The create edge operator possibly adds new edges (properties) to the graph. The name (label) of the edges, as specified by *name*, is the id of a resource of type *rdf:Property* (the id of a property resource). The *subject* and the *object* must have types complying with the domain and the range of the property resource indicated by *name*. If there is already an edge between *subject* and *object* with the label given by *name* then there is no need to create a new edge. Recall that the RDF semantics doesn't allow the presence of two edges that share the same label between the same two nodes.

The *subject* is one node (or singleton collection) in the graph. The *object* can be a collection of nodes. Note that in the above description *object* denotes a node from the input collection. The edges are created between the *subject* node and the *object* node(s). The create edge operator returns the subject node (a collection containing one node). This operation can be generalized after introducing variables in RAL as shown in Subsection 5.1.

Example 12. If $n1$ and $n2$ are the two nodes constructed in Example 11, $n1$ denoting the blank node and $n2$ denoting the literal node, $cedge[name, n1](n2)$ creates an edge labeled $name$ between the nodes $n1$ and $n2$.

Delete node

$$dnode(e : expression)$$

The delete node operation deletes nodes from the graph. The input collection gives the nodes that are removed. The operation returns the empty collection. As a side effect the edges connected to these nodes as subject or object are also deleted.

Example 13. $dnode(\{r2, r3\})$ deletes the nodes $r2$ and $r3$, and all the edges connected to $r2$ or $r3$. For the given model this implies the elimination of the two resources representing paintings and their associated edges.

Delete edge

$$dedge[re\_name, subject](object : expression)$$

The delete edge operation deletes edges from the graph. The edges that are deleted have to start in the *subject* node and to end in one of the nodes from the *object* collection. The name (label) of the edges to be deleted is given by the regular expression re_name, a regular expression over strings. If the *subject* and/or the *object* expressions are empty the edges to be deleted are identified by the remaining input arguments. The operation returns the *subject* input.

Example 14. $dedge(\#, r1)(\{r2, r3\})$ deletes the edges between $r1$ and $r2$, and between $r1$ and $r3$, irrespective of their name. In the concrete example the information that two paintings exemplify the painting technique ($r1$) is removed.

5. Additional RAL features

5.1. Variables

A variable is a substitute for a collection of nodes (possibly) resulting from an evaluation of an algebraic expression. A variable thus serves as a shortcut of such an expression that can be used in more complex algebraic expressions. There are several reasons for introducing variables. First, as we already saw in the definition of the join operator, the join's selection condition may need a reference mechanism for the two operands (input collections). Second, variables can be very useful in expressing complex expressions in which a collection is used repeatedly. The third reason is related to the fact that query languages like RQL give their results in terms of a table that has as columns variables and as rows bindings of these variables. If one would like to use RAL to implement RQL expressions this compatibility feature should be met.

Example 15. $y := \pi[paints](x := r4)$ instantiates x with $r4$ and y with $r2$ and $r3$. If one wants to export these variables, the result will be a table, similar to a table returned by RQL, with two columns x and y, and two rows: the first row contains $r4$ and $r2$ and the second row contains $r4$ and $r3$.

The last reason for having variables is the fact that it has a nice application for the construction operators. If the extracted nodes are bound to variables, these variables can be elegantly used in the construction part of RAL. The create edge operation can be extended by allowing a collection of nodes not only in the *object* part but also in the *subject* part by representing both parts with variables. The semantics of this construction operator is that for each variable binding an edge will be created between the corresponding nodes.

Example 16. Consider the variable bindings from the previous example, $y := \pi[paints](x := r4)$. The expression $cedge[peind, x](y)$ will add two edges with the label *peind* (the French translation of *paints*) to the model, one between $r4$ and $r2$, and one between $r4$ and $r3$.

As shown in the previous example the value of the inner variable (x) is associated with two values of the outer variable (y). The two pairs ($r4$,$r2$) and ($r4$,$r3$) created by the projection operator can be seen as two 2-tuples similar to those from the relational model.

Generalizing this we can say that n-1 nested projections create a set of sets of sets ... of sets (n times) of variable bindings or in other words they generate n-tuples.

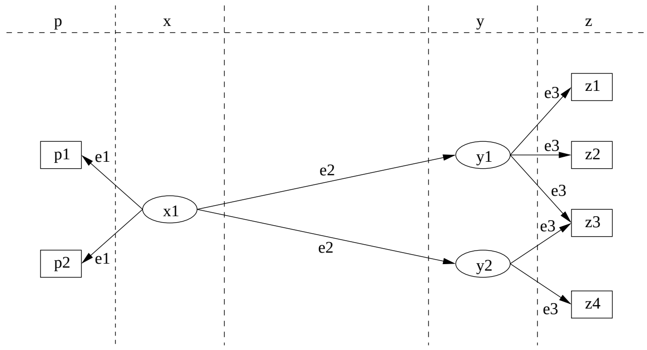

Figure 3. Variable bindings.

Example 17. To illustrate the above consider the tuple bindings for the following expression operating over the RDF graph depicted in Figure 3: $z := \pi[e3](y := \pi[e2](x := x1))$. The resulting bindings are the following 3-tuples: ($x1$,$y1$,$z1$), ($x1$,$y1$,$z2$), ($x1$,$y1$,$z3$), ($x1$,$y2$,$z3$), and ($x1$,$y2$,$z4$).

Note that by generating these tuple bindings we possibly generate duplicates at the variable level (the variable x is bound five times to the same value $x1$ in the above example). These duplicates are removed prior to applying variable bindings as input for an operator in order to assure “duplicate-free” collections on which our operators are closed.

In order to be able to compare results with RDF query languages that use as their output tables of tuples, RAL provides a mechanism to export tuple bindings. This is achieved simply by specifying the variable names participating in the tuple, separated by “,”. For instance x,y,z exports the five tuples from the previous example. Note that if we export only one variable, say x, there will still be five 1-tuples (five times $x1$), i.e. export does not remove duplicates.

So far we discussed only variables which were bound during the multiple application of the projection operator, i.e. they occurred on the same path in the graph. These variables are dependent in the sense that the value of the next variable(s) depends on the binding of the previous ones. There might be, however, variables that do not depend on each other, i.e. they do not appear on the same path in the graph. In case of exporting independent variables export performs a cross product of their bindings.

Example 18. The variable p from $p := \pi[e1](x1)$ is independent from the variables introduced in the previous example. Exporting p, y, z results in the following tuple bindings: ($p1$,$y1$,$z1$), ($p1$,$y1$,$z2$), ($p1$,$y1$,$z3$), ($p1$,$y2$,$z3$), ($p1$,$y2$,$z4$), ($p2$,$y1$,$z1$), ($p2$,$y1$,$z2$), ($p2$,$y1$,$z3$), ($p2$,$y2$,$z3$), and ($p2$,$y2$,$z4$).

5.2. Additional Operators

Sort

$$\Sigma[value\_expression(e)](e : expression)$$

The sort operator orders alphabetically a collection based on a value given by *value_expression*. This *value_expression* is an expression that returns a collection of strings (literals or URI references). The *value_expression* is applied for each node in the input collection and the original nodes are ordered alphabetically based on the computed values.

Note that RAL collections are sets, i.e. they are not ordered. Nevertheless it is useful to be able to output ordered collections, as a last operator to be possibly used in a RAL expression.

Example 19. $\Sigma[\pi[name]](\pi[paints](r4))$ orders alphabetically the resources representing $r4$'s paintings based on their names.

5.3. RQL and RAL

RQL [21] is the most advanced RDF(S) query language to date and RAL was designed taking into consideration RQL's power of expression. RQL path expressions from the `FROM` clause and RQL conditions from the `WHERE` clause can easily be converted in RAL expressions using RAL operators. The vice versa conversion is not always possible as there are RAL expressions (e.g. expressions with construction operators) that are not expressible in RQL. Unlike RAL, RQL is not a closed query language; it takes as input an RDF graph and it returns a table of variable bindings. Since this table does not represent an RDF graph (just values of some variables) it cannot be used again as input for the next query. As a consequence, views are not supported. Nevertheless, RQL offers some degree of nesting queries in the `FROM` and `WHERE` clauses.

Example 20. *Find the name of all painting techniques and the name of the painters who used these techniques.* In RQL this query looks as follows:

```
SELECT Xtn, Zcn
FROM {X:Technique}exemplified_by.painted_by{Z}.cname{Zcn},
     {M}tname{Xtn}
WHERE X=M
```

In our concrete example this query returns two identical rows. The pair *Chiaroscuro*, *Rembrandt* appears twice as a result since there are two paintings ($r2$ and $r3$) that exemplify the *Chiaroscuro* technique and are painted by *Rembrandt* ($r4$).

The following RAL program exports the same variable bindings of Xtn and Zcn as the above RQL query:

$z := \pi[painted\_by](\pi[exemplified\_by](x := \sigma[\pi[rdf{:}type] = Technique]$ $(c))); Xtn := \pi[tname](x); Zcn := \pi[cname](z);\ Xtn, Zcn$

Instead of just outputting variable values in a table-like fashion the construction operators of RAL allow for constructing a full-fledged RDF graph. For instance the following expression connects all painters from the previous query to the techniques they were using by adding a *ptechnique* edge: $cedge[ptechnique, z](x)$.

6. RAL equivalence laws

One of the advantages of using an algebra expression for a query is the ability to rewrite this expression in a form that satisfies certain needs. For example an automatic translator from RQL to RAL can use RAL equivalence laws to rewrite algebra expressions for query optimization purposes.

The proposed set of equivalence laws is inspired by the monad laws [39], and the relational algebra's equivalence laws [37]. In [4] it was shown how relational equivalence laws can be reused (redefined) in an object oriented context.

Law 1 (Left unit). If e_1 is of unit type (singleton collection), i.e. $e_1 = \{n\}$, then

$$e_2(e_1) = e_2(n)$$

Law 2 (Right unit). If e_2 is the identity function, i.e. $e_2(e) = e$, then

$$e_2(e_1) = e_1$$

Law 3 (Empty collection). If e_2 is the empty function, i.e. $e_2(e) = ()$, then

$$e_2(e_1) = ()$$

Law 4 (Decomposition of $\bowtie$).

$$e_1 \bowtie [condition]\ e_2 = \sigma[condition](e_1 \times e_2)$$

Law 5 (Decomposition of π). If *name* is a regular expression that can be decomposed in several regular expressions $name_1$, ... $name_n$ then

$$\pi[name](e) = \pi[name_1](e) \cup ...\pi[name_n](e)$$

Law 6 (Cascading of σ).

$$\sigma[c_1 \wedge ... c_n](e) = \sigma[c_1](...(\sigma[c_n](e))...)$$

Law 7 (Commutativity of σ).

$$\sigma[c_1](\sigma[c_2](e)) = \sigma[c_2](\sigma[c_1](e))$$

Law 8 (Commutativity of σ with π). If the condition c involves solely nodes that have incoming edges named by the regular expression $name$, then

$$\pi[name](\sigma[c(\pi[name])](e)) = \sigma[c](\pi[name](e))$$

Law 9 (Commutativity of σ with $\times$). If the condition c involves solely nodes from e_1, then

$$\sigma[c](e_1 \times e_2) = \sigma[c](e_1) \times e_2$$

Law 10 (Commutativity of σ with $\cup$, $\cap$, $-$). If θ is one of the operators $\cup$, $\cap$, and $-$, then

$$\sigma[c](e_1 \ \theta \ e_2) = \sigma[c](e_1) \ \theta \ \sigma[c](e_2)$$

Law 11 (Commutativity of $\cup$, $\cap$, $\times$). If θ is one of the operators $\cup$, $\cap$, and $\times$ then

$$e_1 \ \theta \ e_2 = e_2 \ \theta \ e_1$$

Law 12 (Commutativity of π with $\times$). If $name$ is a regular expression that can be decomposed in two regular expressions $name_1$ and $name_2$, and if $name_1$ involves solely nodes in e_1, and $name_2$ involves solely nodes in e_2, then

$$\pi[name](e_1 \times e_2) = \pi[name_1](e_1) \times \pi[name_2](e_2)$$

Law 13 (Commutativity of π with $\cup$).

$$\pi[name](e_1 \cup e_2) = \pi[name](e_1) \cup \pi[name](e_2)$$

Law 14 (Associativity of $\cup$, $\cap$, $\times$). If θ is one of the operators $\cup$, $\cap$, and $\times$ then

$$(e_1 \ \theta \ e_2) \ \theta \ e_3 = e_1 \ \theta \ (e_2 \ \theta \ e_3)$$

In order to illustrate the usefulness of the above laws for query optimization we use an example. The query optimization heuristics is based on pushing the selections/projections down as far as possible and applying the most restrictive selections first as it was done similarly in the relational algebra context. The example schema is given in Figure 4.

It is a slightly modified example compared to the one from Figure 2 in the sense that the properties between concepts are replaced by literal (value) properties that function as concept identifier locators. This new example comes from a Web data integration exercise in which different schemas need to be merged "by value". We chose this schema example as it better (compared with the example from Figure 2) illustrates the proposed query optimization.

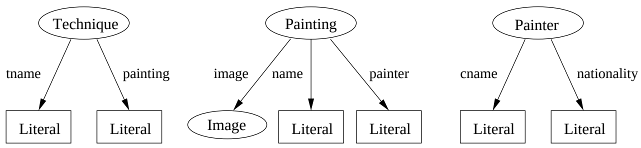

Figure 4. Example schema.

The query under investigation is: *Return in alphabetical order the nationalities of the painters that used the Chiaroscuro painting technique.* A query parser will produce the initial query tree given in Figure 5. In all query trees a represents the collection of all resources in the input model classified under the schema from Figure 4.

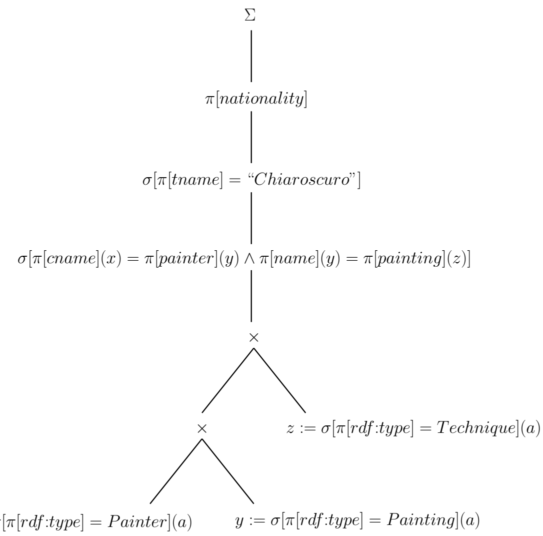

Figure 5. First (initial) query tree.

A query execution module will process a node in a query tree as soon as the operands are available. Such a node will be replaced by the collection that results from executing the node's associated expression. The execution terminates when the root node is processed. The final query result is the collection obtained from processing the root node.

In the example, during the execution of the initial query tree a very large Cartesian product between all painters, paintings, and techniques is generated. By pushing the selections down (using Law 6, Law 7, and Law 9) one can get the query tree in Figure 6.

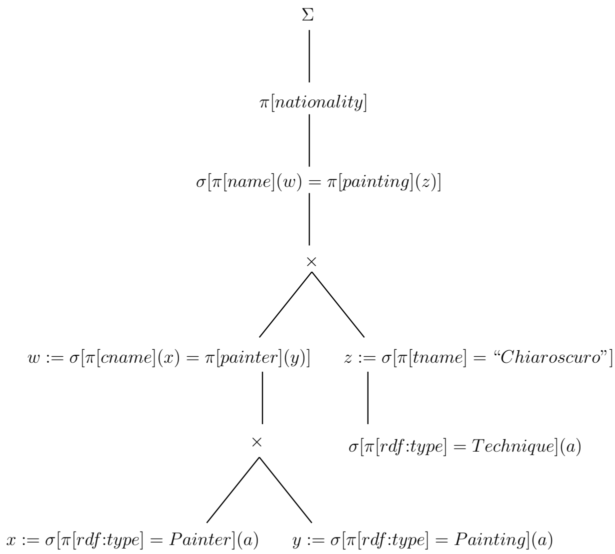

Figure 6. Second query tree.

A further improvement is obtained by applying the most restrictive selections first (using Law 6, Law 7, Law 9, Law 11, and Law 14). The resulting query tree is given in Figure 7. So, with the aid of RAL laws three equivalent query trees were obtained.

In order to better understand why it is more efficient to execute the last query tree, we will give a quantitative dimension to our example. Suppose that the instance of the proposed schema example has 5 painting techniques, 100 painters, and 1000 paintings. Only 100 of all paintings use the Chiaroscuro painting technique. Let's compute now the number of elements generated by the Cartesian products for each query tree. For the first query tree we have 100 x 1000 + 5 x 100 x 1000 = 600,000 elements, for the second query tree 100 x 1000 (painters are matched to their paintings) + 1000 x 1 (paintings are matched to the Chiaroscuro painting technique) = 101,000 elements, and for the last query tree 1 x 1000 (paintings are matched to the Chiaroscuro painting technique) + 100 x 100 (paintings that use the Chiaroscuro technique are matched to their painters) = 11,000 elements. The most efficient to execute is the last query tree as its Cartesian products produce the smallest number of elements.

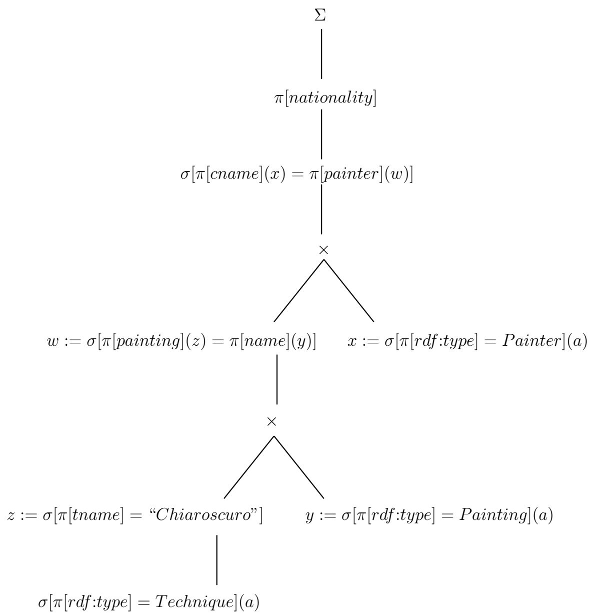

Figure 7. Third query tree.

7. Conclusions

RAL is an RDF algebra defined to support the formal specification of an RDF query language. It presents a set of operations to be used in both the extraction and construction parts of a formally defined RDF query language. It is one of the first RDF algebras developed from a database perspective. Compared with existing RDF query languages, the construction phase is not neglected and is part of the language specification.

As future work we will analyze the expressive power of RAL with respect to existing RDF query languages and its completeness. Comparing the expressive power of RAL to that of other algebras, like relational algebra or object algebra, gives some insight into the real strength of the language, but the true test is the comparison with other existing query languages for RDF.

We would like to further investigate optimization laws that enable algebraic manipulations for query optimization. The lack of order (between resources) in RDF models and RAL collections, as well as the simplicity and composability of RAL operators (similar to the relational algebra ones) seem to foster the definition of RAL optimization laws. A translator from a popular RDF query language (e.g. RQL) to RAL and a RAL engine will enable us to experiment with different aspects of RDF query optimization.

References

1. Allsopp, D., P. Beautement, J. Carson, and M. Kirton: 2002, 'Towards Semantic Interoperability in Agent-based Coalition Command Systems'. In: *The First Semantic Web Working Symposium.*
2. Beckett, D.: 2003a, 'RDF/XML Syntax Specification (Revised)'. W3C Working Draft 23 January 2003. `http://www.w3.org/TR/rdf-syntax-grammar/`.
3. Beckett, D.: 2003b, 'Redland RDF Application Framework'. `http://www.redland.opensource.ac.uk`.
4. Beeri, C. and Y. Kornatzky: 1993, 'Algebraic Optimization of Object-Oriented Query Languages'. *Theoretical Computer Science* **116**(1&2), 59–94.
5. Berglund, A., S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon: 2002, 'XML Path Language (XPath) 2.0'. W3C Working Draft 15 November 2002. `http://www.w3.org/TR/xpath20/`.
6. Berners-Lee, T.: 1998, 'What the Semantic Web can represent'. W3C 1998. `http://www.w3.org/DesignIssues/RDFnot.html`.
7. Berners-Lee, T.: 1999, *Weaving the Web.* Harper San Francisco.
8. Biron, P. V. and A. Malhotra: 2001, 'XML Schema Part 2: Datatypes'. W3C Recommendation 02 May 2001. `http://www.w3.org/TR/xmlschema-2/`.
9. Boag, S., D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon: 2002, 'XQuery 1.0: An XML Query Language'. W3C Working Draft 15 November 2002. `http://www.w3.org/TR/xquery/`.
10. Brickley, D. and R. V. Guha: 2003, 'RDF Vocabulary Description Language 1.0: RDF Schema'. W3C Working Draft 23 January 2003. `http://www.w3.org/TR/rdf-schema/`.
11. Catell, R. G. G., K. D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez: 2000, *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann.
12. Connolly, D., F. van Harmelen, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein: 2001, 'DAML+OIL (March 2001) Reference Description'. W3C Note 18 December 2001. `http://www.w3.org/TR/daml+oil-reference`.
13. Decker, S., D. Brickley, J. Saarela, and J. Angele: 1998, 'A Query and Inference Service for RDF'. In: *The W3C Query Languages Workshop.* `http://www.w3.org/TandS/QL/QL98/pp/queryservice.html`.
14. Decker, S., S. Melnik, F. Van Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann, and I. Horrocks: 2000, 'The Semantic Web: The Roles of XML and RDF'. *IEEE Internet Computing* **4**(5), 63–74.
15. Frasincar, F., G. J. Houben, and C. Pau: 2002, 'XAL: an Algebra for XML Query Optimization'. In: *Database Technologies 2002, Thirteenth Australasian Database Conference*, Vol. 5 of *Conferences in Research and Practice in Information Technology.* pp. 49–56.
16. Grant, J. and D. Beckett: 2003, 'RDF Test Cases'. W3C Working Draft 23 January 2003. `http://www.w3.org/TR/rdf-testcases/`.
17. Guha, R. V.: 2000, 'RdfDB Query Language'. `http://www.guha.com/rdfdb/query.html`.
18. Guha, R. V., O. Lassila, E. Miller, and D. Brickley: 1998, 'Enabling Inferencing'. In: *The W3C Query Languages Workshop.* `http://www.w3.org/TandS/QL/QL98/pp/enabling.html`.
19. Hayes, P.: 2003, 'RDF Semantics'. W3C Working Draft 23 January 2003. `http://www.w3.org/TR/rdf-mt`.

20. Intellidimension Inc: 2002, 'RDFQL Query Language Reference'. http://www.intellidimension.com/RDFGateway/Docs/querying.asp.
21. Karvounarakis, G., V. Christophides, D. Plexousakis, and S. Alexaki: 2001, 'Querying RDF Descriptions for Community Web Portals'. In: *17iemes Journees Bases de Donnees Avancees*. pp. 133–144.
22. Kay, M.: 2002, 'XSL Transformations (XSLT) Version 2.0'. W3C Working Draft 15 November 2002. http://www.w3.org/TR/xslt20/.
23. Klyne, G. and J. J. Carroll: 2003, 'Resource Description Framework (RDF): Concepts and Abstract Syntax'. W3C Working Draft 23 January 2003. http://www.w3.org/TR/rdf-concepts/.
24. Kokkelink, S.: 2001, 'Transforming RDF with RDFPath'. Working Draft. http://zoe.mathematik.Uni-Osnabrueck.DE/QAT/Transform/RDFTransform.pdf.
25. Lassila, O.: 2001, 'Enabling Semantic Web Programming by Integrating RDF and Common Lisp'. In: *The First Semantic Web Working Symposium*. pp. 403–410.
26. Lassila, O. and R. R. Swick: 1999, 'Resource Description Framework (RDF) Model and Syntax Specification'. W3C Recommendation 22 February 1999. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222.
27. Malhotra, A. and N. Sundaresan: 1998, 'RDF Query Specification'. In: *The W3C Query Languages Workshop*. http://www.w3.org/TandS/QL/QL98/pp/rdfquery.html.
28. Marchiori, M. and J. Saarela: 1998, 'Query + Metadata + Logic = Metalog'. http://www.w3.org/TandS/QL/QL98/pp/metalog.html.
29. McBride, B.: 2001, 'Jena: Implementing the RDF Model and Syntax Specification'. In: *Second International Workshop on the Semantic Web*.
30. Melnik, S.: 1999, 'Algebraic Specification for RDF Models'. Working Draft. http://www-diglib.stanford.edu/diglib/ginf/WD/rdf-alg/rdf-alg.pdf.
31. Melnik, S.: 2001, 'RDF API Draft'. http://www-db.stanford.edu/~melnik/rdf/api.html.
32. Miller, L.: 2002, 'Inkling: RDF query using SquishQL'. http://swordfish.rdfweb.org/rdfquery.
33. Prud'hommeaux, E.: 2002, 'Algae HOWTO'. W3C. http://www.w3.org/1999/02/26-modules/User/Algae-HOWTO.html.
34. Seaborne, A.: 2001, 'RDQL - A Data Oriented Query Language for RDF Models'. HP Labs. http://www.hpl.hp.com/semweb/rdql.html.
35. Sintek, M. and S. Decker: 2002, 'TRIPLE - an RDF Query, Inference, and Transformation Language'. In: *The Semantic Web - ISWC 2002, First International Semantic Web Conference*, Vol. 2342 of *Lecture Notes in Computer Science*. pp. 364–378.
36. Thompson, H. S., D. Beech, M. Maloney, and N. Mendelsohn: 2001, 'XML Schema Part 1: Structures'. W3C Recommendation 02 May 2001. http://www.w3.org/TR/xmlschema-1/.
37. Ullman, J. D.: 1989, *Principles of Database and Knowlwdge-Base Systems*, Vol. 1&2. Computer Science Press.
38. van Harmelen, F., J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein: 2003, 'Web Ontology Language (OWL) Reference Version 1.0'. W3C Working Draft 21 February 2003. http://www.w3.org/TR/owl-ref/.
39. Wadler, P.: 1992, 'Comprehending Monads'. *Mathematical Structures in Computer Science* **2**(4), 461–493.