A Semantic-aware RDF Query Algebra

Li Chen Amarnath Gupta M. Erdem Kurul

San Diego Supercomputer Center, University of California at San Diego
9500 Gilman Drive, La Jolla, CA 92093-0505, USA
{lichen|gupta|erdem@sdsc.edu}

Abstract

In this paper, an RDF algebra based on a layered RDF graph model is proposed to incorporate semantics inferencing into query answering. We achieve this by providing a set of interpreting rules to expand the pattern graph specified in a user's query into a fuller one which explicitly captures the required inferencing semantics.

1 Introduction

The Semantic Web was first proposed in [6] as "an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation". Since then, it has attracted a collaborative effort led by W3C with participation from many researchers and industrial partners. The Resource Description Framework (RDF) [5] along with its companion specification RDF Schema (RDFS) [7] have emerged as the basic building block for the semantic web by providing the fundamental mechanisms for describing the semantic relationships between Web resources.

In the RDF model [20], the atomic constructs are statements, which are subject-predicate-object triples describing a set of resources connected via properties. A collection of RDF statements can be intuitively viewed as a graph: resources are nodes and properties that connect them are edges. In addition, the RDF graph structure is enriched with the model-theoretical semantics endorsed by RDFS [16], which defines the inheritance hierarchies of classes and properties.

1.1 Motivation

A basic requirement for the interoperability of RDF-based systems is to not only deal with the lexical and structural properties of RDF statements, but also support the *semantic inference* of such data to discover the complex relationships among described resources.

Many RDF-based software toolkits attempt to meet this demand by applying different methods. Some [25] implement native rule-based inference engines while others [1, 4] solve the problem by plugging in existing description-logic-based reasoners such as Fact [17] and Racer [14]. The inference provided by the systems in the first category is based upon manually added inference rules. No generic inference facilities have been investigated, based on RDF/RDFS semantics, which would apply to any schema instance. Furthermore, understanding the precise semantics of these system proprietary rules is difficult due to the lack of a common formal framework, let alone adapting to them. For the second approach, the limitation is that a third-party reasoner is often treated as a black box and hence no interference or composition with other operations can be easily incorporated. More importantly, these reasoners are designed for statically checking the knowledge base consistency, concept satisfiability and subsumption, etc. It is hence not possible or practical to directly use such reasoners for the dynamic retrieval-oriented query service. For example, Racer [14] provides a functional API for retrieving all individuals that are instances of a given concept. However, it is impossible for Racer to express the query to "find instances x and y which share a common parent".

We see the needs for both querying and inferencing RDF data and advocate in the view of RDF the paradigm of *inferencing as part of query answering* [3], which has begun to gain recognition from researchers in the database community. The view is to incorporate inference, besides structural extraction and manipulation, into query algebra operators.

1.2 Related Work

There are already a number of RDF query proposals including RQL [19], SquishQL [21], TRIPLE [28], RDQL [26], SeQL [8], SPARQL [23], etc. The query capabilities of a variety of RDF language proposals have been compared against each other using the metrics in a recent survey [15]. This metrics is composed of both general and RDF-specific query language desiderata. The general desired query properties include 1)

International Conference on Management of Data
COMAD 2005b, Hyderabad, India, December 20–22, 2005

relational completeness which requires the capabilities of selection, projection, theoretic set operations, etc., 2) operational closure requiring that the results of an operation are again elements of the data model, 3) adequacy which means that the underlying data model is fully exploited, 4) orthogonality which asks for uncorrelated semantics of primitive operators. The RDF-specific properties refer to the capability of querying the graph structure of RDF using constructs like path expression, optional path, and functionalities that exploit the RDF features such as namespace, reification, and entailment.

There also exist efforts in utilizing the research result of querying general graph models for querying RDF. Among these graph data model proposals, GOOD [13] and graphDB [11] are based on object models, Graphlog [9] and G-log [22] are based on logic models and Gram [2] presents a set of algebraic operators. Most of them require the graph data to be compliant with a certain rigid schema, and then base the graph-oriented query operations on complicated combinations of value searches and joins. Since all these proposals are developed before the emergence of RDF, none of them suit to satisfy the RDF-specific properties. With respect to algorithmic aspects of querying and indexing graphs, there is a recent survey on graph searching [27].

Through the study of literature, we observe two weaknesses in current RDF query research. First, an input RDF is more than a simple subject-predicate-object graph structure, it also incorporates RDFS vocabulary (a.k.a. RDF schema [7]) that annotates the class and property types of data resources as well as describes the inheritance hierarchies of these types. Among the implemented RDF query language proposals [15], many of them lack inference (or so-called entailment) capabilities which allow implicit knowledge to be obtained by reasoning over the explicit knowledge. Others have restricted inference capabilities by applying different subsets of entailment rules.

Second, there lacks a comprehensive query algebra for RDF, one like the relational algebra for relational databases, which provides a set of operators with clearly defined semantics as the formal basis for systematic RDF query implementations. To our best knowledge, RAL [10] is the only RDF algebra that has been proposed to date. RAL has three types of operators: extraction, loop, and construction. All operations in RAL have the following form $o[f](x1, x2, \ldots, xn : expression)$, where o is a general symbol for any operator, *expression* is a collection of nodes, and f is a function having as input/output collections of nodes. In other words, the fundamental query data model of RAL is relational node set, and the extraction-based RAL operators are basically relational operators. Therefore, RAL does not encompass adequate capabilities needed for querying the RDF-specific graph structure. For example, path expressions and optional path are not considered in RAL. In addition, RAL largely overlooks the need for inferencing for RDF. For instance, if the domain of a property is of a specific class, RAL cannot be used to infer that a resource that has this property is actually an instance of this class.

1.3 Our Contributions

The contributions of this work include the following.

1. We introduce in Section 3 the notion of "layer" in an RDF graph model based on which queries that require implicit inferencing over RDF schema (hence called schema-aware queries) can be identified and distinguished from those involving only graph operations.

2. We propose (also in Section 3) a new RDF algebra which targets to satisfy both general and RDF-specific query properties, with inference incorporated into query answering.

3. For the pattern-matching-centric algebra operators, we provide a set of interpreting rules (in Section 4) to expand the pattern graph specified in a user's query into a fuller pattern which explicitly represents the needed inference semantics.

4. The proposed matching pattern interpretation module can be incorporated into any RDF query implementation for the inference capability.

To recap, the focus of this work is to design a new RDF algebra and its interpreting rules which turn the implicit inference semantics into explicit pattern matching.

2 Preliminaries

2.1 Graph Data Model for RDF

The underlying structure of any expression in RDF is a collection of triples, each consisting of a *subject*, a *predicate* and an *object*. A set of such triples is called an RDF graph with predicate edges connecting subjects and objects. Formally, the universe of an RDF graph *G, universe(G)*, is a set of *resources* which essentially are anything that can have a *universal resource identifier* (URI). Assume *U, B, L* represent sets of URI references, blank nodes (i.e., anonymous objects), and RDF literals respectively, *universe(G)* is the set of elements of *UBL* (i.e., the set union of *U, B* and *L*) that occur in the triples of *G*.

An RDF model M is a set of triples (statements) each represented by $(s, p, o) \in (U \cup B) \times Pr \times (U \cup B \cup L))$ $(Pr \subseteq U \cup B)$. The subject and object of a triple can be blank nodes. Suppose $G_M = (N, E, l_N, l_E)$ is a graph corresponding to M. Then $l_N : N \rightarrow UBL$,

$l_E : E \rightarrow Pr$. A construction function from M to G_M includes the following steps for each $(s, p, o) \in M$: 1) add nodes n_s, n_o to N (different only if $s \neq o$), 2) assign $l_N(n_s) = s$, $l_N(n_o) = o$, 3) add a directed edge e_p from n_s to n_o into E, and 4) assign $l_E(e_p) = p$.

Subgraph Isomorphism and RDF Entailment. Subgraph isomorphism is to decide if there is a subgraph of one graph which is isomorphic (i.e., existing a one-to-one correspondence mapping between vertices and edges) to another graph. For two RDF graphs G_1 and G_2, a subgraph isomorphic mapping $\mu : G_1 \rightarrow G_2$ preserves the UBL of G_1 in that of G_2, i.e., $\mu(u_1) = u_2$ and $\mu(l_1) = l_2$ for all $u_1 \in U_1$ and $l_1 \in L_1$ (resp. $u_2 \in U_2$, $l_2 \in L_2$), and $\mu(G_1)$ as the collection of all $(\mu(s_1), \mu(p_1), \mu(o_1))$ such that $(s_1, p_1, o_1) \in G_1$ is a subgraph of G_2. Assume G_1 and G_2 are two simple RDF graphs, i.e., those that do not use RDFS vocabulary (introduced next) with a predefined semantics. Then G_1 *entails* G_2, denoted $G_1 \models G_2$, if and only if $G_2 \rightarrow G_1$ [12]. Roughly speaking, the notion of entailment captures information inferencing, in that if $G_1 \models G_2$, the information in G_2 is also (explicitly or implicitly) present in G_1.

2.2 RDFS Vocabulary and Its Interpreted Semantics

RDF's Vocabulary Description Language, also called RDF Schema (RDFS for short) [7], is introduced by W3C to extend the semantics of RDF with class and property descriptions. Specifically, resources may be divided into groups called *classes*. The members of a class are known as *instances* of the class. A resource is declared to be an instances of a class using the property $rdf:type$. The most important classes include $rdfs:Resource, rdfs:Class, rdfs:Literal, rdfs:Datatype, rdfs:XMLLiteral, rdf:Property$. The relationships between subject resources and object resources are called *properties*. The *domain* and *range* of a property describe the classes of resources to which the property applies and leads to respectively. The important properties are $rdf:type, rdfs:domain, rdfs:range, rdfs:subClassOf, rdfs:subPropertyOf$, etc. With the predefined semantics for these reserved words defined in RDFS vocabulary, RDF graphs are no longer simple but support typing and inheritance. For example, Figure 1 shows an example of a set of art resources described by an RDF schema.

For simple RDF graphs, entailment means direct graph mapping. On the other hand, if G_1 and G_2 are two RDF graphs that use RDFS vocabulary with predefined semantics, then the model theory defined rules of entailment [16, 12], as shown below, need to be applied first to infer the implicit information in each graph based on which $G_1 \models G_2$ can be determined.

Rule Set 1: Simple Graphs

$$G \models G' \quad for\ a\ map\ \mu : G' \rightarrow G \tag{1}$$

Rule Set 2: SubClass (sc)

$$(a, type, class) \models (a, sc, a) \tag{2}$$
$$(a, sc, b),\ (b, sc, c) \models (a, sc, c) \tag{3}$$
$$(a, sc, b),\ (x, type, a) \models (x, type, b) \tag{4}$$

Rule Set 3: SubProperty (sp)

$$(a, type, property) \models (a, sp, a) \tag{5}$$
$$(a, sp, b),\ (b, sp, c) \models (a, sp, c) \tag{6}$$
$$(a, sp, b),\ (x, a, y) \models (x, b, y) \tag{7}$$

Rule Set 4: Domain/Range (dom/range)

$$(a, dom, c),\ (x, a, y) \models (x, type, c) \tag{8}$$
$$(a, range, d),\ (x, a, y) \models (y, type, d) \tag{9}$$

Rule set 1 essentially describes the semantics of simple graphs. Subclass rules (corresponding to rules rdfs9, rdfs10 and rdfs11 in [16]) generate the transitive closures of *subclass* $\rightarrow$ *class* and *instance* $\rightarrow$ *class* links. Subproperty rules yield the transitive closure of *subproperty* $\rightarrow$ *property* links and also propagate property values down the subproperty chain (rdfs5, rdfs6 and rdfs7). Domain/Range rules infer resource types based on domain and range scopes (rdfs2 and rdfs3).

Utilizing these entailment rules, more information may be inferred from an RDF graph. For example, by applying rules in rule set 2, $\&r2 \xrightarrow{rdf:type}$ `Artifact` is an inferred fact in Figure 1 from $\&r2 \xrightarrow{rdf:type}$ `Painting` and `Painting` $\xrightarrow{rdfs:subClassOf}$ `Artifact`. In case that the original knowledge base represented by the RDF graph is incomplete, e.g., the link $\&r2 \xrightarrow{rdf:type}$ `Painting` is missing, then we can infer this fact from $\&r1 \xrightarrow{paints} \&r2$ and range(`paints`)=`Painting`.

RDF Graph Closure and Reduction. Given an input RDF graph G, its *closure* and *reduction* refer to respectively the fullest and the minimal structures that are both equivalent to G in representing path connectivity. The former is computed by applying the specified entailment rules in an exhaustive manner forwardly, whereas the latter is derived by an inverse procedure. Usually, an RDF store tends to keep only the triples in a reduced RDF graph and delay the needed closure computation until runtime. The reason is that a full closure computation may generate explosively many new triples which unnecessarily aggravate the memory consumption for a unforeseeable application workload.

3 Layered Graph Algebra for RDF

3.1 "Layers" in RDF Graph

We consider an RDF graph has three "layers". **Layer 1** consists of only those subject-predicate-object triples

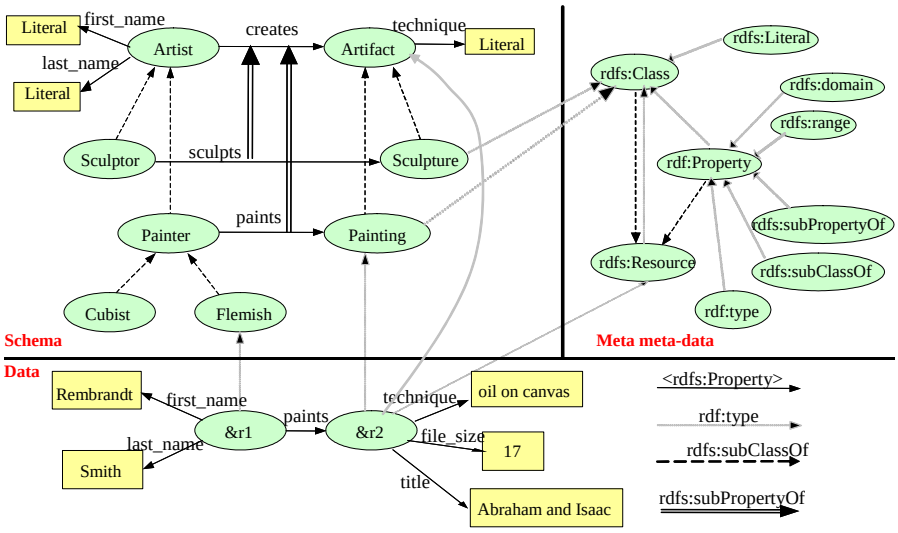

Figure 1: RDF Graph Example

that do not involve RDFS vocabulary. This is referred to as a schema-less barebone *data layer*. **Layer 2** is the *schema layer*, containing the schema information, i.e., application-specific classes and properties, of the triples at the data layer. In this layer, RDFS vocabulary including rdf:type, rdfs:domain, rdfs:range, rdfs:subClassOf, rdfs:subPropertyOf are involved in describing the meta information, such as the application class to which an instance resource belongs, the property type of a predicate and its corresponding domain and range, the class and property hierarchies, etc. **Layer 3** is the *meta meta-data layer*, composed of the type declarations of application classes and properties as rdfs:Class and rdf:Property respectively, links from instance resources or application classes to rdfs:Resource, and relationships among RDF classes and properties (i.e., rdfs:Class, rdf:Property, rdfs:Resource, rdfs:Literal, rdfs:XMLLiteral, etc.). For instance, an application class `Painting` in Figure 1 is an application class and hence an instance of rdfs:Class, which is a subclass of rdfs:Resource, and `creates`, `sculpts` and `paints` are instances of rdf:Property, etc.

Both layer 2 and layer 3 involve RDFS vocabulary, e.g., using rdf:type to describe the class of a resource. However, the domain and range values of the property rdf:type for triples (in appearance of edges in RDF graphs) in these two layers are different. Specifically, the rdf:type edges in layer 2 can only point to application-specific classes and properties but not RDF classes and properties. For example, triple $\&r2 \xrightarrow{rdf:type}$ `Painting` belongs to layer 2, whereas triples `Painting` $\xrightarrow{rdf:type} rdfs:Class$ and $\&r2 \xrightarrow{rdf:type} rdfs:Resource$ belong to layer 3. Also, the domain of the rdf:type properties in layer 2 cannot be RDF classes or properties.

3.2 Schema-aware RDF Queries

Corresponding to the layered graph model, we classify the RDF queries into three categories: 1) data layer queries that use no RDFS vocabulary and query only against the data layer resources, 2) schema-aware queries that enquire information in the schema layer, and 3) meta-data queries that are concerned about the meta meta-data layer data. Schema awareness is one of the most important requirements for RDF queries. In the layered model, it translates directly to the awareness of layer 2 of the RDF graph.

Among existing RDF query proposals, RQL and SeRQL support entailment in RDF natively and SeRQL even allows to distinguish between subclasses and direct subclasses. For example, an RQL query can be used to express "$subClassOf($`Artist`$)$" or "$select\ \$C1, \$C2\ from \{\$C1\}$`creates`$\{\$C2\}$", where `Artist` is an application class and variables $\$C1, \$C2$ are capitalized to indicate that they range over application classes. Suppose that the second query is posted against the RDF graph in Figure 1, then the answer pairs for ($C1,$C2) include not only (`Artist,Artifact`), but also (`Sculptor,Sculpture`) and (`Painter,Painting`) since `sculpts` and `paints` are subPropertyOf `creates`. However, to our best knowledge, little work has been done to systematically incorporate such inferencing into the general graph-oriented querying model to enhance schema awareness for RDF queries. We hence propose a new RDF algebra to overcome this.

3.3 Graph-based RDF Algebra – LAGAR

Inspired from TAX [18], a tree algebra for XML that exploits a set of operators centered around a pattern tree structure for identifying the subset of nodes of interest in the input tree collection, we correspond-

ingly propose a **LAyered Graph Algebra for RDF** (**LAGAR** for short). Since the RDF model is based on graphs rather than trees, we first introduce the notion of *pattern graph* which is central to the semantics of many operators in LAGAR.

Definition 3.1 (Pattern Graph) *Formally, a pattern graph P is a graph with its labeled edges and nodes constrained by formulas. I.e., $P = (N, E, F_N, l_E)$ such that $F_N : N \rightarrow pred(\theta, N_G)$, $l_E : E \rightarrow re(E_G)$, where N_G and E_G are two sets composed of UBL and Pr ($Pr \subseteq UB$) of the data graph G respectively, wildcard *, and RDFS vocabulary; θ is a comparison operator (e.g., $=$, $\neq$, $\geq$) against string constants or variables, and $re(E_G)$ denotes a regular expression over E_G vocabulary. More generally, F_N may be a boolean combination of predicates.*

Like the pattern tree for TAX, the notion of graph pattern here provides a simple yet intuitive specification of nodes and their connecting relationships of interest. This matching function $h : P \rightarrow G$ is a total mapping (i.e., embedding) from nodes and edges of P to nodes and paths of G such that: 1) h preserves the structure of P, i.e., whenever $(u \xrightarrow{re} v)$ is an edge in P, $h(v)$ is a node that is connected to $h(u)$ via a path with the concatenated edge labels satisfying re; 2) $h(v)$ satisfies the formula F_N for all $v \in N$.

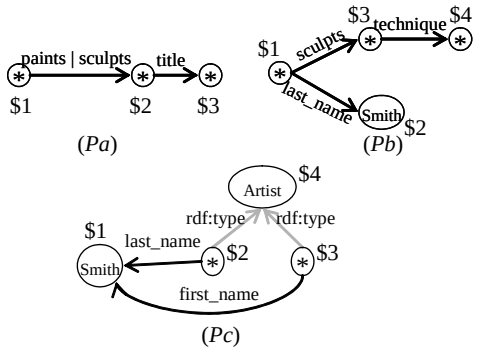

Figure 2: Pattern Graph Examples

For example, Figure 2 gives three pattern graphs for the RDF in Figure 1. The (path-like) pattern graph Pa in Figure 2 requests for triples whose subjects are connected to objects via either "`paints`" or "`sculpts`" predicates, and also the adjacent triples that have "`title`" predicates. In the (twig-like) pattern graph Pb, a pattern node specifies the constraint that requires the corresponding objects must be the RDF literal "`Smith`". The (dag-like) pattern graph Pc is interested in sculptors whose `first_names` or `last_names` are "`Smith`". It involves RDFS vocabulary and hence is schema-aware.

The Operators. To satisfy both general and RDF-specific desired properties, we design our LAGAR algebra which can be classified into four main categories of operators as below.

Pattern-matching operators. In this category, operators are oriented to structural selection and extraction by employing pattern graphs.

Construction operators. These operators are designed to facilitate the result graph construction for RDF queries by providing means for creating and inserting new nodes/edges and manipulating the extracted structures.

Graph-set operators. Operators in this category take as input a collection of graphs (a collection of nodes or edges are the extreme cases) and perform set-theoretical operations, although some may not have exactly identical semantics as their relational counterparts.

Functional operators. We classify into this category the operations of aggregation, and the *apply* function borrowed from the functional language context that can be used to apply a user-defined function to each member of the input collection.

A fundamental property of LAGAR is that each operator in the algebra accepts a collection of RDF graphs as input and returns graphs as the result. This property ensures computation closure and composability. Generally, a LAGAR operator is denoted as $op_{pa}^{out}(C)$, where op is the specific operator, C denotes the input graph collection, pa is the input parameter adornments, and out is the produced output. Implicitly, the *apply* function is used to apply op_{pa}^{out} to each graph G_i in C, and the union of all outputs of $op_{pa}^{out}(G_i)$ composes the output of $op_{pa}^{out}(C)$.

We show in Table 1 the important LAGAR operators in each of the four categories. Among the pattern matching operators in the first category, selection and projection appear to have similar input parameter adornments, i.e., P for the pattern graph and SL (resp. PL) for the list of node types to be retained in the output structure. However, selection and projection have different semantics that respectively correspond to the two *return semantics* for matching P against a graph $G_i \in C$. Specifically, the output of $\sigma_P^{SL}(C)$ is a collection of *witness graphs* of P, one per embedding of P into G_i ($G_i \in C$) restricted to the set of nodes which correspond to the pattern nodes appearing in the adornment list SL. For the pattern matching of $\sigma_P^{SL}(C)$ that results in multiple witness graphs which share a common node u in G_i as the image of a node in P, u is repeated in the output of this selection.

In contrast, the output of $\pi_P^{PL}(C)$ includes the images of all embeddings of P into G_i ($G_i \in C$) that are confined by the pattern nodes in PL, however emphasizing that the precedence order among the retained nodes in the original structure is preserved. That is, for any two nodes u and v in the output of $\pi_P^{PL}(C)$, whenever u precedes v in the sense that there is a path from u to v in $G_i \in C$, u proceeds v in the output.

Description	Notation	Semantic
	Pattern-matching Operators	
Select	$\sigma_P^{SL}(C)$	return witness graphs of P in G_i ($\forall G_i \in C$), nodes restricted to SL.
Project	$\pi_P^{PL}(C)$	same as above, with nodes preserving their precedence order in G_i.
Product	$\times(C_1, C_2)$	pair each graph G_i in C_1 with each graph G_j in C_2 and connect their *root* nodes (i.e., those that have only outgoing edges) to a common new root via the property edges of a null type.
Join	$\bowtie_P^{SL}(C_1, C_2)$	$\{G_{ij} \mid G_{ij} = \sigma_P^{SL}(\times(G_i, G_j), \forall G_i \in C_1, \forall G_j \in C_2)\}$ (P is to be matched against $\times(G_i, G_j)$, at least one f in the F_N of P is $\$v = \$u$ ($\$v$ matches nodes in G_i, and $\$u$ matches nodes in G_j).
Outer Join	$=\!\!\bowtie_P^{SL}(C_1, C_2)$	P_i and P_j are the two parts in P that are matched against each G_i in C_1 and each G_j in C_2 respectively, if no witness graph G_j' obtained from $\sigma_{P_j}^{SL_j}(G_j)$ satisfies the join condition $\$v = \$u$, then output just $\sigma_{P_i}^{SL_i}(G_i)$; otherwise, output $\sigma_P^{SL}(\times(G_i, G_j))$.
	Graph-set Operators	
Groupby	$\gamma_{gf}^{GL}(C)$	partition graphs in C by the grouping function gf, then within each partition connect all roots of the graphs being projected by GL list.
Distinct	$\delta(C)$	keep only the unique graphs in C up to isomorphism, G_1, G_2 are isomorphic if exist maps $\mu 1, \mu 2$ s.t. $\mu 1(G_1) = G_2$ and $\mu 2(G_2) = G_1$.
Union	$\cup(C_1, C_2)$	the set theoretical union of C_1 and C_2.
Merge	$+(C_1, C_2)$	$\delta(\cup(G_1, G_2))$, i.e., blank nodes in G_2 that are identical to those in G_1 are removed and then the remaining G_2 is union-ed with G_1.
Intersect	$\cap(C_1, C_2)$	the set theoretical intersection of C_1 and C_2.
Difference	$-(C_1, C_2)$	the set theoretical difference of C_1 and C_2.
	Construction Operators	
Insert Node	$\odot(C, G_i, c, id)$	create and insert into G_i a instance of class c which has an id being either a uri, blank, or a literal (c is $rdfs{:}Literal$ in the last case).
Insert Edge	$\oslash(C, G_i, id_s, p, id_o)$	create a property edge labeled p to connect subject id_s to object id_o.
Change NValue	$\rho(C, G_i, id, l)$	change in G_i the literal value of a node with id to be new value l.
Change EProp	$\beta(C, G_i, id_s, id_o, p)$	change in G_i the old property type on the edge $id_s \longrightarrow id_o$ to be p.
Delete Node	$\ominus(C, G_i, id)$	remove the node with id in G_i.
Delete Edge	$\boxslash(C, G_i, id_s, id_o)$	remove from G_i the edge that connects subject id_s to object id_o.
	Functional Operators	
Aggregation	$\mathcal{A}_{af,us}(C)$	apply the aggregate function af and insert the generated value(s) back at the position according to the update specification us.
Apply	$\Diamond_f(C)$	apply function f (user-defined or an operator) to each G_i in C.

Table 1: LAGAR Operators

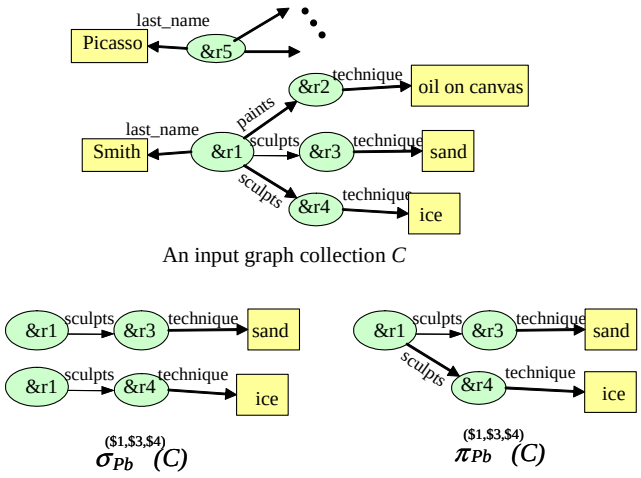

Figure 3: Selection vs. Projection

This implies that the semantics of projection may be regarded as eliminating nodes other than specified in PL from G_i rather than repeatedly output nodes in each different witness graphs. Such a difference in the output structures of $\sigma_P^{SL}(C)$ and $\pi_P^{SL}(C)$ is illustrated in Figure 3 for the same pattern graph P, graph collection C, and output list SL.

For aggregation operations, the purpose is to map collections of values to aggregate or summary values. Common aggregate functions are MIN, MAX, COUNT, SUM, etc. Due to the property of "closeness", the output of an aggregation operator is still graphs, but are updated with the computed summary values based on the *update specification us*. The example us can be simply a node identifier id, function $lowest{-}common{-}ancestor(id_1, id_2)$, etc.

Similar to TAX [18], grouping and aggregation are separate orthogonal operators in LAGAR. The rationale for this separation arises from the difference in

nature of the two operators – grouping for RDF graphs involves partition of the graphs and restructuring the output, whereas aggregation is mainly value-based instead of structure-based.

It is worthy to note that the operators listed in Table 1 are not meant to be complete nor exclusive. They however represent a clean core set of operations which are primitive, orthogonal, composable, and intuitive for manipulating the graph-structured RDF data enriched with RDF vocabulary. It should be rather easy to derive other operators that are of interest to RDF data from this core set.

Comparison of LAGAR with Other Algebraic Approaches. Here we particularly compare our proposed LAGAR with RAL [10] and some representative graph data model proposals [13, 11, 2].

In the design of RAL [10], the operators are defined to work on collections of "nodes", which form the RAL query data model. Edges are not considered the "first-class" citizens and they can only be accessed via projecting on the specified property type of a collection of nodes. Furthermore, RAL follows the approach of binding variables to graph nodes, and then manipulating the use of these variables through looping constructs where needed. A natural implementation of a loop-based RAL query expression corresponds to a "nested-loops" execution plan. In comparison, the guideline for designing our LAGAR is to avoid the functional-programming-style recursion where possible, and thus devise a bulk manipulation algebra that enables the access and processing of RDF elements in a "set-at-a-time" fashion. For queries involving recursion such as introduced by the *apply* function, an implementation of LAGAR can provide an explicit support for iteration.

Among the existing well-known graph data models, GOOD [13] represents the scheme as well as the object instances using a graph and express data manipulations by graph transformations, while GraphDB [11] and Gram [2] propose explicit graph data models. In terms of query algebra design, Gram [2] uses a data model where walks (paths) are the basic objects and walk expressions are regular expressions with alternating sequences of node and edge types. The data model exploited by GraphDB [11] is a collection of object classes divided in: simple classes (simple objects that represent nodes), link classes (links between nodes that represent edges) and path classes (representing several paths in the database).

The commonality among these existing graph data models is that a collection of graphs are transformed in the first phase of query processing into a collection of walks or other component units of graphs but not graphs themselves. Then most of the query operations can be performed in a way similar to the relational algebra to further manipulate the derived collections. The ultimate output graphs can be constructed in one final step. Such a paradigm of query processing may incur large overhead accumulated by repeated construction and deconstruction steps. It would hence not be appropriate or efficient to employ these graph query algebras to handle queries that semantically require many of such round-trips or to deal with data with high connectivity such as RDF. In contrast, our LAGAR algebra handles collections of graphs directly. Like TAX for XML data, LAGAR avoids the aforementioned problems but needs to tackle the issue of heterogeneity. The notion of *pattern graph* provides the central semantics of LAGAR operators and it offers the needed standardization over a heterogeneous set. Hence, LAGAR is a "proper" algebra that applies to a heterogeneous collection of graphs.

Another important feature of LAGAR is its schema-awareness and the capability of inferencing over the class and property hierarchies of RDF data. We will show in the next two sections that LAGAR allows efficient inference based on graph-pattern-based matching algorithms. The limitation of our LAGAR algebra is however primarily its inadequacy in exploiting *all* elements of the underlying data model. For example, the concepts of RDF containers, reification, namespace, etc., have not yet been accommodated in LARGA. In this sense, additional operators will be needed to handle these model elements. In this paper, we confine our discussions to the proposed LAGAR operators and focus on the inference-enabling pattern graph interpreting process for the pattern-matching operators.

4 Inference-enabling Pattern Graph Interpretation

As defined earlier in Section 3.3, the basic elements of a pattern graph P are nodes constrained by formulas comparing against $UBL \cup \{*\} \cup C_{rdfs}$ (C_{rdfs} represents the RDFS classes) and edges against $re(Pr \cup \{*\} \cup P_{rdfs})$ (P_{rdfs} denotes the RDFS properties). For a given P and an RDF graph G that both use RDFS vocabulary, pattern matching is no longer simply embedding P directly into G. We may need to utilize the entailment rules (see Section 2.2) to derive a "full" pattern graph which explicitly represents the matching semantics. We call this the interpretation process of a pattern graph and describe it next.

4.1 Pattern Graph Interpretation Module

To incorporate inferencing into schema-aware query answering, we propose a pattern graph interpretation module to be plugged in any RDF query implementation system. The interfacing of this module with other parts of an RDF query implementation is illustrated in Figure 4.

After the input query is parsed, the pattern graphs involved in it are either extracted directly if the RDF query system is implemented based on graph pattern

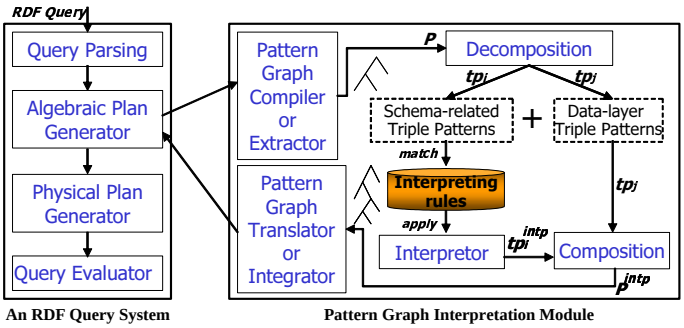

Figure 4: Interfacing Pattern Graph Interpretation Module with an RDF Query System

matching, or otherwise compiled from the native syntax or operator and constructed inside of our interpretation module. Each pattern graph P is then decomposed into the component triples tp_i each composed of two nodes and an edge possibly with variables. These triples are divided into two groups, i.e., *schema-related* and *data-layer*, based on whether schema layer information or RDFS vocabulary is involved. Triples in either group can be further classified into a fixed set of atomic patterns called **triplet graphs**. In other words, triplet graphs are categorized triple components of a pattern graph.

Each triplet graph can be interpreted using a corresponding interpreting rule. For a schema-related triplet graph, applying its interpreting rule may result in a "blown" pattern graph. Finally, the interpreted triple patterns are merged together to compose a pattern graph P^{intp} (^{intp} denotes "interpreted") which is ready to be matched against the input RDF graph. If the RDF query system is *not* implemented based on graph pattern matching, then the interpreted "full" pattern graph P^{intp} is translated back into the native syntax or operator.

For example, below we show how a pattern-matching LAGAR operator is used to express a schema-aware RQL query and how it is further decomposed and classified into several triplet graphs.

***Example* 1**

RQL query:
select x, $X, y
from {x : $X} paints {y}
and {y} title "Sunflower"

LAGAR operator: $\sigma_P^{x,\$X,y}(C)$, *where P is a pattern graph somewhat like $P(a)$ in Figure 2 (with no alternative label "sculpts" though on the left direct edge) and C is the input RDF graph collection.*

Triplet Graphs: The pattern graph P used above is composed of two triplet graphs: one is $(?x : ?X, a, ?y)$ (the only non-variable a here stands for property "paints") and the other is $(?y, b, z)$ whose only variable is y while b and z are constants. The possible types of triplet graphs will be introduced next.

4.2 Interpreting Data-layer Triplet Graphs

Table 2 shows eight atomic data-layer triplet graphs. They differ in the number and positions of *variables* (i.e., those prefixed by '?', a symbol borrowed from the RQL syntax) in them. A pattern node, either at the subject or object position, of a triplet graph is associated with a variable only if it appears in the selection list SL of σ or the projection list PL of π. Data nodes matching these variables are to be extracted, similar to the use of '$' in XQuery for variable bindings.

4.3 Interpreting Triplet Graphs

In a schema-related triple pattern, we use capital letters $A, B, .., Z$ to denote the pattern nodes and edges that are to be matched with schema-layer data, while denoting those that are to be matched with data-layer resources by non-capital letters $a, b, .., z$. For example, compared to the data-layer triple pattern (x, a, y), $(?x : A, a, y)$ is a schema-related triple pattern additionally requiring that the matching subjects must be instances of class A. All schema-related triple patterns can be classified into the atomic triplet graphs listed in Table 3. They differ in the number and positions of class or property variables (i.e., capitalized variables).

The interpretation of a schema-related triple pattern needs to utilize the predefined semantics of RDFS vocabulary by reversely applying the appropriate entailment rules mentioned in Section 2.2. For example, suppose a resource $\&r_i$ is the subject of a triple that matches $(?x, a, y)$, we need to check if $\&r_i$ is an instances of class A, either directly or transitively. In other words, if there are triples $\&r_i \xrightarrow{rdf:type} C$ and $C \xrightarrow{rdfs:subClassOf} A$ in the input RDF graph, then $\&r_i \xrightarrow{rdf:type} A$ according to the SubClass entailment rule set. Below we give the interpreting process for this example triple pattern.

$$(?x{:}A, a, y) \xrightarrow{':'= typeOf} (?x, a, y),\ (?x, type, A)$$
$$\xrightarrow{sc\ rule\ 4} (?x, a, y),\ (?x, type, ?C),\ (?C, sc, A)$$
$$\xrightarrow{sc\ rule\ 2,3} (?x, a, y),\ (?x, type, ?C),\ (?C, (sc)*, A)$$

In the above example, the final interpreted term is a

Data-layer Triplet Graph Semantics	
(x, a, y):	all-constant triplet graph is considered as a boolean pattern testing if such a triple exists in the input RDF graph.
$(?x, a, y)$:	extracting the subject nodes of the triples that match (x, a, y).
$(x, ?a, y)$:	extracting the predicates of the triples that match (x, a, y).
$(x, a, ?y)$:	extracting the object nodes of the triples that match (x, a, y).
$(?x, ?a, y)$:	extracting the subject and property pairs of the matching triples.
$(?x, a, ?y)$:	extracting the subject and object node pairs of the matching triples.
$(x, ?a, ?y)$:	extracting the predicate and object pairs of the matching triples.
$(?x, ?a, ?y)$:	extracting all three elements of the matching triples.

Table 2: Semantics for Data-layer Triplet Graphs

Interpreting Rules for Schema-related Triplet Graphs	
All-constant Boolean Triplet Graphs	
(A, a, y)	$\rightarrow (?x, a, y), (?x, type, ?C), (?C, (sc)*, A)$
(x, P, y)	$\rightarrow (x, ?a, y), (?a, (sp)*, P)$
(x, a, B)	$\rightarrow (x, a, ?y), (?y, type, ?D), (?D, (sc)*, B)$
(A, P, y)	$\rightarrow (?x, ?a, y), (?x, type, ?C), (?C, (sc)*, A), (?a, (sp)*, P)$
(A, a, B)	$\rightarrow (?x, a, ?y), (?x, type, ?C), (?C, (sc)*, A), (?y, type, ?D), (?D, (sc)*, B)$
(x, P, B)	$\rightarrow (x, ?a, ?y), (?y, type, ?D), (?D, (sc)*, B), (?a, (sp)*, P)$
(A, P, B)	$\rightarrow (?x, ?a, ?y), (?x, type, ?C), (?C, (sc)*, A), (?a, (sp)*, P), (?y, type, ?D), (?D, (sc)*, B)$
Schema-data Variable Based Triplet Graphs	
$(?C, a, y)$	$\rightarrow (?x, a, y), (?x, type, ?C), (?C, (sc)*, ?A), (a, dom, ?A)$
$(x, ?P, y)$	$\rightarrow (x, ?a, y), (?a, (sp)*, ?P)$
$(x, a, ?D)$	$\rightarrow (x, a, ?y), (?y, type, ?D), (?D, (sc)*, ?B), (a, range, ?B)$
$(?C, ?P, y)$	$\rightarrow (y, type, ?D), (?D, (sc)*, ?B), (?P, range, ?B), (?P, dom, ?A), (?C, (sc)*, ?A)$
$(?C, a, ?D)$	$\rightarrow (a, dom, ?A), (a, range, ?B), (?C, (sc)*, ?A), (?D, (sc)*, ?B)$
$(x, ?P, ?D)$	$\rightarrow (x, type, ?C), (?C, (sc)*, ?A), (?P, dom, ?A), (?P, range, ?B), (?D, (sc)*, ?B)$
$(?C, ?P, ?D)$	$\rightarrow (?P, type, rdfs{:}Property), (?P, dom, ?A), (?P, range, ?B), (?C, (sc)*, ?A), (?D, (sc)*, ?B)$

Table 3: Interpreting Schema-related Triplet Graphs

conjunction form of mixed types of triplet graphs. For example, $(?x, a, y)$ is a data-layer triplet graph while the other two are schema-related. In $(?C, (sc)*, A)$, pattern node $?C$ is connected to class A via an edge labeled by $(sc)*$. This means that the class bindings of $?C$ are transitive subclasses of A since *sc* represents $rdfs{:}subclassOf$. Repeated variables in the conjunction form indicate the joint points where the corresponding triplet graphs meet. In this case, the expanded pattern graph P^{intp} that captures the matching semantics of $(?x{:}A, a, y)$ is shown in Figure 5.

Besides containing pattern nodes and edges that are to be matched with the application classes and properties in the RDF graph, schema-related triplet graphs may also involve RDFS vocabulary such as *type* and *sc* (stand for *rdf:type* and *rdfs:subClassOf* respectively). In this case, interpreting them needs to not only reversely apply the entailment rules, as illustrated by the example below, but also explore related schema information. For instance, $(?C, a, y)$ means $(*{:}?C, a, y)$ and it requests to extract the class(es) of the subjects in the triples that match $(*, a, y)$. No entailment rule is appropriate to be applied to $(?C, a, y)$, nor can it be directly matched against the RDF graph since that the class variable $?C$ is specified to be connected to an instance resource y via a predicate a rather than RDF vocabulary. We interpret this by making use of the schema knowledge which confines the bindings of $?A$ within the domain of property a.

$$(?C, a, y) \quad \stackrel{intro\ ?x}{\longrightarrow} (?x{:}?C, a, y) \quad \stackrel{use\ dom}{\longrightarrow} (?x, a, y), (?x, type, ?C), (?C, (sc)*, ?A), (a, dom, ?A)$$

In the first step of this interpretation, a pattern node variable $?x$ is introduced to explicitly indicate that its binding set cannot be empty. In other words, $(?C, a, y)$ requires that there exists at least one instance s of any class binding of $?C$ such that (s, a, y) is a triple in the data layer of the input RDF graph. Note that x is the new variable name and the label of its corresponding pattern node is actually '*'. This is the case for any newly introduced variable. Also note that only the bindings of the original variable(s) in each rule head are to be returned, while those of the newly introduced variables in each interpreting rule are intermediate results that lead to deriving the required variable bindings.

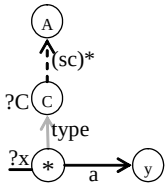

Figure 5: P^{intp} of $(?x{:}C, a, y)$, only bindings of the underlined $?x$ are returned

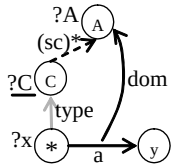

Figure 6: P^{intp} of $(?C, a, y)$, only bindings of the underlined $?C$ are returned

In the second step, the interpretation explores the schema knowledge which relates $?C$ to the domain value A of property a via the subClassOf links. The final interpreted form (corresponding to the first row in the second subgroup of rules in Table 3) is shown in Figure 6 and it is ready to be matched against the input RDF graph.

Similarly, the interpreting rules for the other schema-related triplet graphs are given in Table 3. For a schema-related triple pattern that contains non-capital variable(s) to be matched with data-layer resources, we find that its interpreted term resembles that for the triplet graph obtained by removing such variable(s). For instance, $(?x, ?P, y)$ is interpreted as $(?x, ?a, y)$, $(?a, (sp)*, ?P)$, which only differs from the interpreted form of $(x, ?P, y)$ by the '?'-prefixed x. We hence do not elaborate on the interpreting rules for such triplet graphs in Table 3.

4.4 Properties of P^{intp}

Feasibility. As can be observed from Table 3, an important property possessed by the P^{intp} of each atomic triplet graph is *feasibility*. The notion of *feasibility* comes from the context of answering query using views with binding patterns [24] where it describes the property that the complete answer to a query is computable based on the given binding pattern views. Here, all the interpreted conjunctive forms in Table 3 present similar binding patterns, namely, at least one subgoal has a free variable and two bound variables while the others each has two free variables and one bound variable. For example, in the interpreting rule $(x, ?P, y) \rightarrow (x, ?a, y)$, $(?a, (sp)*, ?P)$, the binding pattern for the first subgoal is $(x^b, ?a^f, y^b)$. Thus the bindings of $?a$ can be derived from this subgoal, which makes $?a$ a bound variable in other subgoal(s), e.g., $(?a^b, type^b, ?Q^f)$. This way, free variables in some subgoals become "bound" in others. Even for those schema-related triplet graphs that contain non-capital variables such as $(?x, ?P, y)$, each P^{intp} has at least one subgoal with no less than one bound variable. This guarantees a feasible order in executing the matching of P^{intp}.

Minimality. Another important property is the *minimality* of each P^{intp}. This refers to the fact that each P^{intp} captures the inferred, redundant-free structural constraints for the corresponding triplet graph. For example, suppose that the first interpreting rule in Table 3 is changed to $(A, a, y) \longrightarrow (?x, a, y)$, $(?x, type, ?C)$, $(?C, (sc)*, A)$, $(a, dom, ?B)$, $(A, (sc)*, ?B)$. The last two added constraints are redundant since the newly introduced variable $?B$ is unnecessary for checking if (direct or indirect) instances of A (i.e., bindings of $?x$) exist to satisfy $(?x, a, y)$.

The redundancy issue that may arise from composing the interpreted forms of the component triples of a pattern graph P is worthy noting. To check and remove redundancies in the merged pattern graph P^{intp}, the approaches of variable unification and graph matching may be employed. We'd like to also point out that, although each interpreting rule "blows up" a triplet graph by a certain constant number, the size of P^{intp} is in linear of the size of P.

4.5 Discussions

In general, the complexity of our proposed LAGAR operator set is $ExpTime$ since its expressive power is equivalent to that of $\mathcal{SHIQ}$, a variation logic under the Description Logic (DL) category. Compared to the DL-based reasoning approach, the algebraic approach we adopt for inferencing is different from the former in the following aspects.

First, the major task of a DL-based reasoner is to statically check the concept coherence, satisfiability and subsumption; while the inferencing capability that concerns us comes from the on-demand schema-aware queries. Second, the two approaches differ in their implementation strategies. A modern DL reasoning system is often based on an optimized tableaux algorithm. Our algebraic approach first analyzes whether the query's resolution demands inference rules to be applied based on the identification and extraction of the "schema-aware" parts. If such parts exist, it classifies them into different triplet graphs and then applies the corresponding interpretation rules to turn them into full matching patterns. Note that the inference engine in a DL system assumes that the facts are known beforehand, which is however, in contradiction with the on-demand retrieval-oriented query paradigm.

To summarize, the inference need arising from the schema-aware queries is turned into pattern matching operations using our approach and the complexity of it is now closely related to the graph database search efficiency.

5 Conclusion

In sum, we addressed in this paper the need for considering inferencing as part of schema-aware RDF query answering. We proposed a layered RDF graph model and built on top of it an RDF algebra which exploits the model theory defined entailment rules and schema knowledge for semantics inferencing. We are currently still at the early phase of designing and implementing a full-fledged RDF query algebra. In the future work, we will analyze the optimization opportunities in a greater depth and seek for solutions for further improvement.

Acknowledgements. This work is supported by NIH Human Brain Project Award No. 5RO1DC03192, NSF ITR Grant EIA-0205061 and NIH Grant P41 RR0088605 for NBCR.

References

[1] Protege: An open source ontology editor tool. http://protege.stanford.edu.

[2] B. Amann and M. Scholl. Gram: A graph data model and query language. In *European Conference on Hypertext*, pages 201–211, 1992.

[3] K. Anyanwu and A. Sheth. ρ-queries: Enabling querying for semantic associations on the semantic web. In *World Wide Web Conference (WWW), Budapest, Hungary*, 2003.

[4] S. Bechhofer and G. Ng. Oiled ontology editor. http://oiled.man.ac.uk.

[5] D. Beckett. Rdf/xml syntax specification. http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210, Feb. 2004.

[6] T. Berners-Lee. Cwm - closed world machine. http://www.w3.org/2000/10/swap/doc/cwm.html, 2000.

[7] D. Brickley and R.V. Guha. Rdf vocabulary description language 1.0: Rdf schema. http://www.w3.org/TR/2004/REC-rdf-schema-20040210, Feb. 2004.

[8] J. Broeskstra and A. Kampman. Serql: A second generation rdf query language. In *SWAD-Europe Workshop on Semantic Web Storage and Retrieval, Amsterdam, Netherlands*, Nov. 2004.

[9] M.P. Consens and A.O. Mendelzon. Graphlog: a visual fromalism for real life recursion. In *Symposium on Principles of Database Systems (PODS), Nashville, Tennessee*, pages 404–416, 1990.

[10] F. Frasincar, G. Houben, R. Vdovjak, and P. Barna. Ral: an algebra for querying rdf. In *Web Information Systems Engineering (WISE), Singapore*, 2002.

[11] R. H. Guting. GraphDB: Modeling and querying graphs in databases. In *International Conference on Very Large Data Bases (VLDB), Santiago, Chile*, pages 297–308, 1994.

[12] C. Gutierrez, C. A. Hurtado, and A. O. Mendelzon. Foundations of semantic web databases. In *Symposium on Principles of Database Systems (PODS), Paris, France*, page 95106, 2004.

[13] Marc Gyssens, Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, June 1994.

[14] V. Haarslev, R. Mller, and M. Wessel. Racer - a semantic middleware. http://www.sts.tu-harburg.de/ r.f.moeller/racer.

[15] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A comparison of rdf query languages. http://www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query/rdfquery.pdf, 2004.

[16] P. Hayes. Rdf semantics. http://www.w3.org/TR/2004/REC-rdf-mt-20040210, Feb. 2004.

[17] I. Horrocks. The fact system. http://www.cs.man.ac.uk/ horrocks/FaCT.

[18] H. V. Jagadish, L. V. Lakshmanan, D. Srivastava, and K. Thompson. Tax: A tree algebra for xml. In *Intl. Workshop on Database Programming Languages (DBPL)*, pages 149–164, 2001.

[19] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Schol. Rql: A declarative query language for rdf. In *World Wide Web Conference (WWW), Hawaii, USA*, May 2002.

[20] G. Klyne and J. Carroll. Resource description framework (rdf): Concepts and abstract data model. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210, Feb. 2004.

[21] L. Miller, A. Seaborne, and A. Reggiori. Three implementations of squishql, a simple rdf query language. In *International Semantic Web Conference (ISWC)*, pages 399–403, 2002.

[22] J. Paredaens, P. Peelman, and L. Tanca. G-log: A graph-based query language. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):436–453, June 1995.

[23] E. Prud'hommeaux and A. Seaborne. Sparql query language for rdf. http://www.w3.org/TR/rdf-sparql-query, Apr. 2005.

[24] A. Rajaraman, Y. Sagiv, and J. Ullman. Answering queries using templates with binding patterns. In *Symposium on Principles of Database Systems (PODS), San Jose, California*, pages 105–112, 1995.

[25] HP Labs Semantic Web Research. Jena a semantic web framework for java. http://jena.sourceforge.net.

[26] A. Seaborne. A query language for rdf. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109, 2004.

[27] D. Shasha, J. Wang, and R. Giugno. Algorithmics and applications of tree and graph serarching. In *Symposium on Principles of Database Systems (PODS), Madison, WI*, pages 39–52, 2002.

[28] M. Sintek and S. Decker. Triple - an rdf query, inference and transformation language. In *Deductive Databases and Knowledge Management (DDLP)*, 2001.