An efficient SQL-based querying method to RDF schemata

Maciej Falkowski[1], **Czesław Jędrzejek**[1]

Abstract: Applications based on knowledge engineering require operations on semantic data. Traditionally, in this area relational data are transformed to a form expected by a reasoning system (usually RDF). In this work a method is presented that transforms a query to a RDF data structure (i.e. SPARQL) to a SQL query executed by RDBMS. This could lead to significant increase to efficiency of reasoning. Our approach is general but does not use ontologies nor Description Logic. Transforming the query from a graph model to a relational model allows introducing rule constructors, which is not possible in the current SPARQL specification.

Keywords: semantic data, graph query, RDF, relational database

1. Introduction

Dealing with data structures occurs on two levels: first on the level of a data model (schema) and second on possible database states. Queries take into account data schema and give answers based on states (instantiated models) (Ramakrishnan, 2002). A major problem of knowledge engineering (where applications based on knowledge databases require operations on semantic data) is that currently prevalent relational data tables lack explicit semantics, although operations on them (queries) are very efficient. One can still realize reasoning by using a semantic query over relational data but a form of the query is not straightforward.

A use semantic data structures, which usually are represented in RDF format (Klyne, 2004) results in smaller query efficiency. To circumvent this (Chong et.al, 2004) implemented an extension to RDBMS, based on a table function infrastructure, which allows rewriting table functions with a SQL query. This extension introduces RDF_MATCH table function, nevertheless processing of a query does not require any additional language run-time system other than the SQL engine. However, their method still requires data in a RDF form. A use of RDF and SPARQL as a query language to RDF structured data is often criticized (Melton, 2006) , claiming that efficiency of such a procedure is low, and the same effect can be achieved using SQL. However, although it is possible to propose a SQL query that is semantically equivalent to a SPARQL query (or more general - a graph-based query), there are situations when SQL queries are much more awkward, harder to construct and less readable. So it is tempting to keep asking queries based on graphs, but answer them using a SQL engine. In this work a method is presented that transforms a query to a RDF data structure (i.e. SPARQL (Prud'hommeaux,

[1] Institute of Control and Information Engineering, Poznan University of Technology, Piotrowo 3, 60-965 Poznan, Poland
e-mail: {maciej.falkowski,jedrzeje}@put.poznan.pl

2007)) to a SQL query executed by RDBMS. It is also shown, that transforming a query from a graph model to a relational model allows introducing constraint constructors, which are not possible in current SPARQL specification. As such, the method facilitates use of reasoning systems.

The paper is organized as follows. Section 2 reviews relevant data structures, mainly RDF, and a query language SPARQL supporting the RDF language. Section 3 discusses a graph matching problem. In Section 4 we present an architecture of a reasoning system using a SQL query to semantic data. In Section 5 an algorithm of transforming graph query to SQL query, based on a mapping between predicate labels and relational table columns, is shown. Section 6 gives the conclusions and future work prospects.

2. Semantic data structures

Most common representation of data used in reasoning is in a form of Resource Description Framework. The base element of the RDF model is the triple: a resource (the subject) is linked to another resource or literal (the object) through an arc labeled with a third resource (the predicate). We will say that <subject> has a property <predicate> valued by <object>. For the sake of a RDF- model relational transformation we introduce the following notation. A relational database table represents an entity set and each row represents an instantiation of the entity. Table name corresponds to a RDF subject (with a proper identifier - a primary key). Names of columns are predicates and values of columns in a tuple are objects. An example is presented in Table 1.

Table 1. Table Persons

Persons					
ID	First name	Last name	PESEL	Born	Place of living
1	Matthew	Black	81070102357	01.07.1981	Warsaw

Entity Person can be represented in the form shown in Fig. 1.

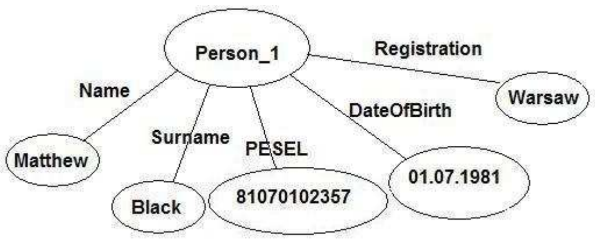

Figure 1. Entity Person in RDF representation

In current data management environment most structured data resides in relational databases. Semantic data has to be extracted from text or prepared from relational data. Unfortunately, permanent data transformation from relational model to RDF model is expensive, also because data synchronization and security methods, offered by contemporary RDBMS, need to be changed. This suggests usefulness of having systems with an on-demand format transformation. There are several systems, such as D2RQ (Bizer et al, 2006), that using an appropriate mapping generate RDF data from relational data "on the fly". Another approach, although of different application value is to transform RDF structured data into a relational model. A trivial transformation is to create a three-column table to represent object, subject and predicate, respectively. Such a relational schema of RDF model is often used to build RDF repositories on top of the RDBMS systems. RDF lacks relationships between properties and other resources. Based on RDF more formal languages, with richer expressive power, have been proposed such as RDF Schema (RDFS) and OWL. Expressive power and decidability of various formalisms and their combinations is an important research topic. Most widely used are systems combining Description Logic (a basis for OWL DL) enhanced by rule extensions. An example of such a language is SWRL (Semantic Web Rules Language) (Horrocks et al., 2004).

3. Basic RDF graph definitions

Our formalism is similar to the one used in (Flesca, Furfaro, Greco, 2006), but specialized to RDF graphs and with use of rules to transform them rather than grammars.

3.1. Data graph, query graph, graph matching and graph derivation

Let Γ be a set of all possible labels, Γ_{URI} a set of labels which are URI, Γ_L a set of labels which are literals, Γ_V a set of labels, which denote variables, Γ_U a set of labels which do not include literals, and $\Gamma_N = \Gamma_U \cup \Gamma_L$ a set of labels which include literals.

A basic element of a graph, a triple, can be defined as:

$$t = (s, p, o) | s, p \in \Gamma_U, o \in \Gamma_N \qquad (1)$$

A triple consists of three elements: subject, predicate and object, of which the first two (s, p) cannot be labeled with literals. Elements s, o are vertices of a graph, and p is an arc between them. The components of triple t will be denoted by S(t), P(t) and O(t) respectively for subject, predicate and object of the triple. Note that in a RDF model it is not allowed to have graphs with nodes not connected with any other node, and that the basic element of graph is a triple; not nodes themselves nor arcs. A graph G can be then defined as a set of triples t, G = t. In the following, for a graph X notation X = t_X is used, where t_X denotes a set of triples of graph X.

Graphs differ according to allowed label sets used to constitute graph's triples. A graph made of triples where $\Gamma_U = \Gamma_{URI}$ (and thus $\Gamma_N = \Gamma_{URI} \cup \Gamma_L$) is called data graph, and a graph made of triples where $\Gamma_U = \Gamma_{URI} \cup \Gamma_V$ (and thus $\Gamma_N =$

$\Gamma_{URI} \cup \Gamma_V \cup \Gamma_L$) is called a query graph. Data graphs consist of triples in which all nodes and arcs are URI or literals, and query graphs allow also using variables. In the following, variables are denoted by strings preceded by a '?'.

3.2. Answering graph queries

Answering a query, described by a query graph Q, over data described by a data graph D is a process of identifying such sub-structures of D which satisfiy a property defined by Q. For this, we define a function which maps a query graph on a given data graph, and we define a new entity, called a mapping pair, consisting of a query graph and a mapping function which associates a query graph with a sub graph of the data graph.

3.2.1. Definition 1 - labels matching and mapping

Let γ, γ' $\in \Gamma$ be two labels, where $\Gamma = \Gamma_{URI} \cup \Gamma_V \cup \Gamma_L$. Such two labels match, if there exists a mapping function $\zeta : \Gamma \rightarrow \Gamma$, such that:

$$\gamma, \gamma' \notin \Gamma_V \Rightarrow \gamma = \gamma', \zeta(\gamma) = \gamma, \zeta(\gamma') = \gamma' \quad \text{or,} \tag{2}$$

$$\gamma \in \Gamma_V \Rightarrow \zeta(\gamma) = \gamma' \quad \text{or,} \tag{3}$$

$$\gamma' \in \Gamma_V \Rightarrow \zeta(\gamma') = \gamma \tag{4}$$

Two labels match if they are not variables and are equal (2), or at least one of them is a variable (3, 4). In the first case, a mapping function ζ maps labels to themselves, and in the second case ζ maps a variable label to another (constant or variable) label. Matching of two labels γ, γ' by a function ζ will be denoted in the following by:

$$\text{label\_match}(\zeta, \gamma, \gamma') \tag{5}$$

3.2.2. Definition 2 - triples matching:

Let t, t' be triples constituted over labels $\Gamma = \Gamma_{URI} \cup \Gamma_V \cup \Gamma_L$. Triple t matches triple t' if a mapping function ζ exists, such that:

$$\text{label\_match}(\zeta, S(t), S(t')) \tag{6}$$

$$\text{label\_match}(\zeta, P(t), P(t')) \tag{7}$$

$$\text{label\_match}(\zeta, O(t), O(t')). \tag{8}$$

Two triples match if there exists such a function ζ, which maps corresponding labels of these triples - subject labels (6), predicate labels (7) and object labels (8). If corresponding labels of the matching triples are constants (URI or literal), then they must be equal for a match to be successful; and if at least one of them is a variable, then mapping from that variable to another label must be the same in the scope of the triple (for all occurrences of that variable in the triple). Matching of two triples t, t' by a function ζ will be denoted in the following by:

$$\text{triples\_match}(\zeta, t, t'). \tag{9}$$

3.2.3. Definition 3 - graph matching

Let Q = t_q be a query graph, D = t_q be a data graph. A mapping from Q to D exists if a label mapping function ζ exists, such that: for each triple tq $\in$ t_q a matching triple td $\in$ t_d exists that:

$$\text{triples\_match}(\zeta, tq, td). \tag{10}$$

A graph Q can be mapped to a graph D if there exists such a mapping function ζ for which all triples in Q can be matched to triples in D (10). Note that the same function ζ is used to match all triples in a query graph, thus the same variable labels, which occur in different triples, are all mapped to the same label.

A mapping pair M_D over graph D consist of a query graph Q a and mapping function ζ, M_D = (Q, ζ). If a mapping pair exists for a given data graph and a query graph, then the answer for that query over that data exists. It is possible that for a given query graph and a data graph multiple answers exist. Every answer maps every element (triple) of a query to element of the data; constant elements of the query are mapped to exactly the same elements in data graph, and thus the most interesting part of mapping is variable mapping - every variable is mapped to a specific, concrete value, and that mappings can differ between answers. On the basis of mapping pair M_D, it is easy to create an answer graph Q^A - a graph of the same structure as the query graph Q, but with variable labels replaced by function ζ , in a way that it forms a sub graph of the data graph. An example of such a mapping and resulting graphs is shown in Fig. 2.

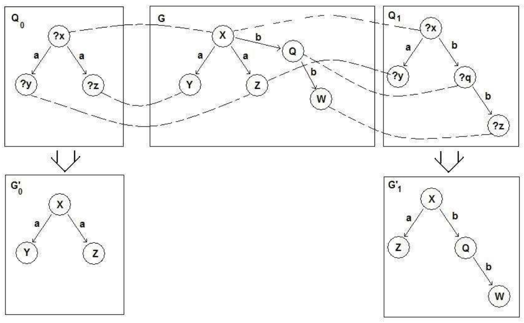

Figure 2. An example of matching graphs.

Here query graphs (Q_0 and Q_1) are mapped to a data graph D, resulting in answer (data) graphs Q_0^A and Q_1^A. Mapping functions ζ in above cases are:

$$\zeta_0 = \{?x : X, ?y : Y, ?z : Z\} \tag{11}$$

$$\zeta_1 = \{?x : X, ?y : Z, ?q : Q, ?z : W\}. \tag{12}$$

3.3. Definition 4 - query graphs transforming

A query graph can be transformed according to a given set of rules. A rule r is an entity which consists of a precondition (body) B and a postcondition (head)

H, r = (B, H). A precondition B is a query type graph, in which variable labels are allowed. A postcondition H is a single triple, also with variable labels allowed. H(r) denotes a head of a rule r, and B(r) denotes a body of a rule r. A rule can be matched to a triple. A rule r matches triple t if H(r) matches t. H(r) and t are single triples with variables can appear, and are matched if a matching function exists, such that triple_match(ζ, H(r), t) (9) is true. This is equivaent to Prolog unification operation.

Given a set of rules R = r, triples t_q of a query graph Q can contain separate sets of terminal triples T_t and non-terminal triples T_n.

$$T_n = \{t | t \in t_q, \exists r \in R.\text{triple\_match}(\zeta, H(r), t)\} \tag{13}$$

$$T_n = \{t | t \in t_q, \neg\exists r \in R.\text{triple\_match}(\zeta, H(r), t)\} \tag{14}$$

The set $T_n \in t_q$ consists of triples for which at least one rule from rule set can be matched. $T_t = t_q \ T_n$ and consists of triples for which no rule can be matched.

A query graph which contains at least one non-terminal triple is called a non-terminal graph. A query graph which contains no non-terminal triples is called a terminal graph.

A non-terminal query graph Q (source graph) can be transformed according to rules to a target graph Q‘ if a non-terminal triple t $\in t_q$ is matched to a rule r by a function ζ. The target graph Q‘ is:

$$Q‘ = (Q \setminus t) \cup \zeta(B(r)) \tag{15}$$

A target graph consists of triples from a source graph, except the matched triple, and of triples of a rule body with labels mapped according to a function ζ. For a given source graph it is possible to find multiple mapping functions ζ, which map different rules to different triples. Target graphs can be terminal or non-terminal, and in the second case can be further transformed. Thus, for a given non-terminal query graph, a transformation tree can be deducted, with the original graph as the root, non-terminal, intermediate graphs as nodes and terminal graphs as leaves. Child nodes of such a tree are parent nodes transformed using (15) according to a particular rule. A single rule can produce multiple descendant nodes when applied to different triples of a parent node, and the parent node can be transformed by multiple rules. A set of child nodes is composed of all possible transformations of parent node.

3.4. Use of the formalism

Formalism introduced in Section 3 serves finding patterns of data. Patterns have a form of query graphs (not necessarily of a tree form as for XQuery search) the nodes of which can be variables or constants, and edges represent predicates. An atomic part of a graph is a triple. To define a pattern one can use predicates obtained from columns of tables of relational schemas. In addition one can create new predicates out of exiting ones. Obtained rules consist of precondition and a conclusion. A precondition may consist of several triples whereas a conclusion is a triple. For example, a predicate hasGrandfather may be created by twofold use of a predicate hasFather as seen in Fig. 3.

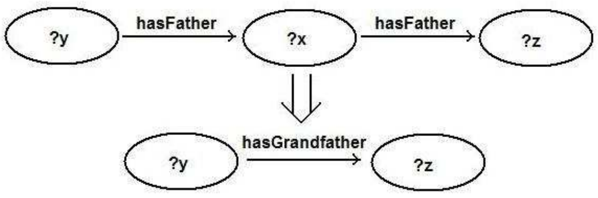

Figure 3. An example of a graph transforming rule

4. An architecture of a reasoning system using SQL query to semantic data

A traditional reasoning system based on rules and utilizing relational data usually consists of the following elements, shown in Fig. 4.

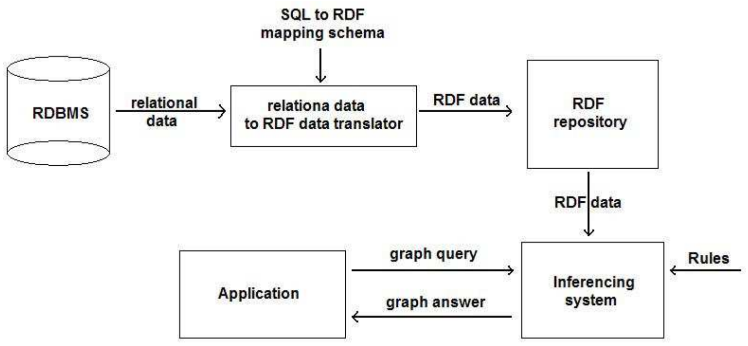

Figure 4. A traditional reasoning system architecture

The central part is an inferencing subsystem such as Racer Pro, Pellet, or Jena. Data come from a repository usually populated by ontologies, native RDF data (not shown in Fig. 4), and virtual RDF data created by a subsystem such as D2RQ. This data is queried with use of set of rules by an application. Inferencing system provides an answer to a controlling application. A flow of data is complex. Relational data have to be (possible virtually) transformed to a RDF format and stored. After this step semantics of information does not change, but many advantages of the RDBMS efficiency are lost (such as B-trees and indices). As a result, time needed to answer queries may be significantly longer than equivalent queries to relational data.

The proposed reasoning system architecture is shown in Fig. 5.

Functions of a reasoning system have been divided into 2 subsystems. The first one takes a graph query (such as SPARQL) and a set of rules, but not RDF data. Using backward reasoning the first subsystem generates all elementary (terminal) graph queries, which contain only predicates directly mapped to database columns.

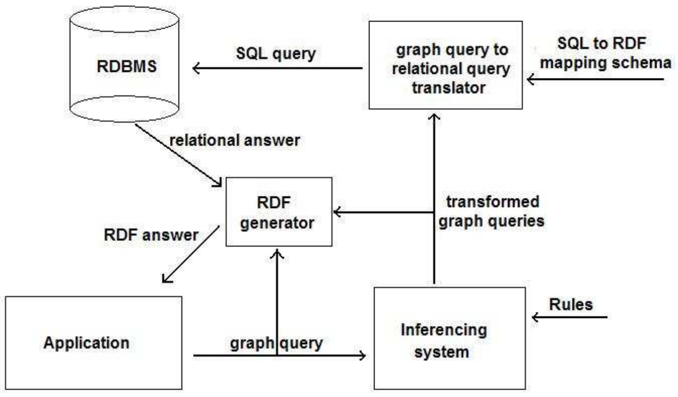

Figure 5. An architecture of reasoning proposed in this work

The second subsystem translates graph queries to SQL queries and joins them. All known efficient mechanisms developed for RDBMS such as index optimization or materialized views may be employed. Answers are in a form of tuples that in sequence can be transformed into RDF data returned to controlling application. Reasoning subsystem is optional, and if a query graph is a terminal graph it can be omitted.

5. Query graph to SQL query translation algorithm

5.1. The algorithm

Definition 5 - predicate labels to a table.columns mapping function m

To translate a query graph into a SQL query it is necessary to know a mapping between predicate labels and relational table columns. This mapping can be done by a function m : $\Gamma_{URI} \rightarrow$ tab.col. M maps URI labels to a pair table.column. By Tab(m) we denote the table part of this pair, and by Col(m) we denote the column part of this pair.

Query graph to SQL query translation algorithm

Input: query graph Q, predicate labels to column names mapping function m
Output: SQL query

1. Group all triples t $\in$ Q into groups g_i , such that every triple in given group has the same subject:

$$g_i = \{t | t \in Q, \forall t, t' \in g_i . S(t) = S(t')\}$$

2. With every group g_i associate a relational table T(g_i), such that predicates of all triples within group have a corresponding column in the same table

T(g_i), based on a mapping function m:

$$T(g_i) = table|\forall t, t' \in g_i.\exists(Tab(m(P(t))), Tab(m(P(t')))).table = Tab(m(P(t))) = Tab(m(P(t')))$$

3. Assign unique table alias a_i to every group g_i.
4. For every triple t ∈ Q:
   (a) Assign mapping string W to every subject that is a variable:
   $S(t) \in \Gamma_V \Rightarrow W(S(t)) = a_i.ID$
   and mapping string W' to every subject that is not a variable:
   $S(t) \notin \Gamma_V \Rightarrow W'(S(t)) = a_i.ID$
   where a_i is the alias of a group where t belongs
   (b) Assign a mapping string E to every object that is variable:
   $O(t) \in \Gamma_V \Rightarrow E(O(t)) = a_i.T(g_i)$
   and mapping string E' to every object that is not variable: $O(t) \notin \Gamma_V \Rightarrow E'(O(t)) = a_i.T(g_i)$
   where g_i is the group where t belongs and a_i is it's alias
5. Add every element of W and E to SELECT part of query.
6. Add every group alias a_i to FROM part of query.
7. For every triple t | $S(t) \notin \Gamma_V$ add WHERE element of form:
   W'(S(t)) = S(t)
8. For every triple t | $O(t) \notin \Gamma_V$ add WHERE element of form:
   E'(S(t)) = O(t) if $S(t) \notin \Gamma_V$ or
   E(S(t)) = O(t) if $S(t) \in \Gamma_V$
9. For every pair t, t' that S(t) = O(t') add WHERE element of form:
   W(S(t)) = E(O(t'))
10. The final result is a composition of elements SELECT, FROM and WHERE.

5.2. Example of the algorithm

For illustration we demonstrate a query to entities described in Tab. 2.

In natural language a query would read: "Find persons whose first name is John and whose father has an address in the city of Poznań". In a relational model this information can be stored in two tables: Persons and Addresses (Tab. 2).

Mappings of properties into column names has the following form:
T = {hasFather: Father's Id, hasName: Name, hasAddress: AddressId, Street: Street, City: City }

The first step of the algorithm is to divide a graph into groups with the same subject. Each group represents one entity, and thus one tuple in an appropriate table. For the graph in Fig. 8 this gives three groups illustrated in Fig. 7. The second

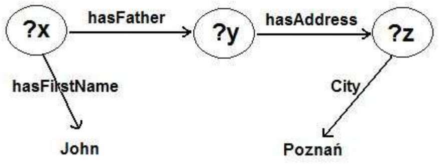

Figure 6. Example of a query

Table 2. Table Persons and Addresses

Persons				
ID	First name	Last name	Father's Id	Address Id

Addresses			
ID	Postal Code	Street	City

step is to map appropriate tables to groups. Group 'a' has two predicates: hasFather and hasFirstName, which both can be mapped into Persons table, columns Father's Id and FirstName.

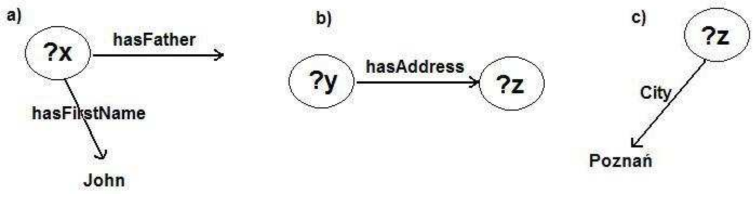

Figure 7. Query groups for a query in Fig. 6

5.3. Detailed procedure

Similar to group 'a' we process other groups, resulting in the following:
group a - Table Persons, alias O1,
group b - Table Persons, alias O2,
group c - Table Addresses, alias A1.

Finally, triple elements are transformed into SQL statements for each group, e.g.:

group a:

- ?x : O1.ID
- ?y : O1. Father'sId

- 'Maciej' : O1.FirstName

The SELECT part is created based on variable mappings and consist of SQL statements corresponding to every variable in the graph query. If a variable appears both as a subject and an object, the statement in the subject form is selected:

SELECT O1.ID, O2.ID, A1.ID

The part FROM of a query is formed using tables and aliases associated with particular groups; each entity (group) has its alias, and in this example it will be:

FROM Persons O1, Persons O2, Addresses A1

Conditions arise from mappings related to constants and variables in two roles: both as a subject and as an object. For every constant node of a query graph, and for every relation between entities, a WHERE element is added:

WHERE O1. First name = 'Maciej' AND A1.City = 'Poznań' AND
O1.Father'sId = O2.ID AND O2.AddressId = A1.ID

This completes a transformation of a graph query to a SQL query to relational data.

6. Summary

In this paper we demonstrated a method that translates a conjunctive query to a graph into a SQL query to relational data of the same semantics as the original (virtual) RDF data. In the process of this translation, use of rules is allowed to transform original query to terminal queries. This method divides reasoning with rules into two steps - building queries according to rules (with backward reasoning) and answering that queries. The last step exploits power of efficient data extracting of RDBMS. The method can be an important element of a reasoning engine, improving scalability and retaining all efficient mechanisms of RDBMS. As such the method is particularly suitable for web applications, which often require near tealtime response. In many cases a graph-model query provides better readability than a direct SQL query, and our method allows asking readable and easy to formulate graph queries. In a future work we will concentrate on extending graph queries with additional constructs, like cardinality constraints on number of occurrences of some entity in given relation, which we find useful and possible to translate to SQL queries. We will also attempt to employ an external rule reasoner, such as Prolog, to efficiently transform graph queries.

This work has been supported by the Polish Ministry of Science and Higher Education, Polish Technological Security Platform grant 0014/R/2/T00/06/02 and the 45-087/07/BW grant. The authors are grateful to prof. T. Pankowski for many discussions.

References

Bizer, C., Cyganiak, R., Garbers, J. and Maresch, O. (2006) D2RQ V0.5 - Treating Non-RDF Relational Databases as Virtual RDF Graphs *User Manual and Language Specification.*
`http://www.wiwiss.fu-berlin.de/suhl/bizer/d2rq/spec/2006103`.

Chong, E., Das, S., Eadon, G. and Srinivasan, J. (2005) An efficient SQL-based RDF querying scheme *Proceedings of the 31st international conference on Very large data bases*, 1117-1127.

Flesca, S., Furfaro, F. and Greco, S. (2006) A graph grammars based framework for querying graph-like data *Data Knowledge Engineering* **59**, 3, 652–680.

Horrocks, I. et al. (2004) SWRL: A Semantic Web Rule Language Combining OWL and RuleML *W3C Member Submission.*
`www.w3.org/Submission/SWRL`.

Klyne, G. and Carroll, J. (2004) Resource Description Framework (RDF): Concepts and Abstract Syntax *W3C Recommendation.*
`http://www.w3.org/TR/rdf-concepts/`.

Melton, J. (2006) SQL, XQuery, and SPARQL: What's Wrong With This Picture? *XTech 2006: "Building Web 2.0"*

Prud'hommeaux, E. and Seaborne, A. (2007) SPARQL Query Language for RDF *W3C Working Draft.*
`http://www.w3.org/TR/rdf-sparql-query/`.

Ramakrishnan, R. and Gehrke, J. (2002) *Database Management Systems, 3rd Ed..* McGraw-Hill, New York.