Control and Cybernetics

vol. **38** (2009) No. 1

An efficient SQL-based querying method to RDF schemata*

by

Maciej Falkowski and Czesław Jędrzejek

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

Abstract: Applications based on knowledge engineering require operations on semantic data. Modern systems have to deal with large data volumes and the challenge is to process and search these data effectively. One of the aspects of efficiency is the ability to query those data with respect to their semantics. In this area an RDF data model and a SPARQL query language gained the highest popularity and are de facto standards. The problem is that most data in modern systems are stored in relational databases and have no formal and precisely expressed semantics, although operations on those data are fast and scalable. Traditionally, in this area relational data are transformed to a form expected by reasoning and querying systems (usually RDF based). In this work a method of query rewriting is presented that translates a query to an RDF data structure (i.e. SPARQL query) to a SQL query executed by RDBMS. Transforming queries instead of data has many advantages and might lead to significant increase of data extraction efficiency.

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

1. Introduction

Dealing with data structures occurs on two levels: firstly, on the level of a data model (schema) and secondly, on the possible database states with a database schema. Queries take into account a data schema and give answers based on states (instantiated models) (Ramakrishnan and Gehrke, 2002). A major problem of knowledge engineering (where applications based on knowledge databases require operations on semantic data) is that currently prevalent relational databases lack explicit semantics. Querying such data requires

*Submitted: June 2008; Accepted: October 2008.

knowledge about their structure and meaning, which is not easy to express with existing relational database tools. The lack of standards in this area makes it also difficult to share the knowledge about schema semantics with interested parties. There appears a serious problem when data from different sources have to be integrated within one application. Similarly named relational tables and columns can possibly contain different data, for example a column 'date' in a table 'products' can contain production date or expiration date. It is clear that table and column names are not sufficient for expressing the data semantics. Another drawback of the relational representation is that SQL queries are hard to formulate compared to SPARQL queries. As we show in Section 2, it is possible to construct an SQL query to relational data equivalent to a given SPARQL query to data in an RDF format. The later is easier to understand and analyze. Switching from the relational to the semantic data model and queries can bring many advantages. First, in this area RDF and ontologies can become a lingua franca of the data representation. The key concept here is similar to the federated databases concept (Heimbigner and McLeod, 1985), where multiple heterogenous databases are integrated with the use of a special, mediated schema. Here we map existing relational schemas from different databases to classes and terms constituting a common and shared domain ontology (an equivalent of the mediated schema). This would guarantee compatibility on the data representation level (RDF instead of many relational schemas) and the semantic level (common terminology to express data meaning). Querying of such unified data would be much easier. Another benefit of using a semantically-oriented data model, such as RDF, is the ability to reason about queries and get more complete answers. Reasoning systems can exploit domain knowledge, contained in an ontology in the form of a concept (classes and relations) hierarchy. For example, knowing that motorcycles and cars are vehicles, one can ask a query about vehicles and get answers containing motorcycles and cars also. This ability is very powerful and enabling an RDF access to relational data is one step towards Semantic Web and a potential revolution in the domain of search engines. Another related issue is the storage of native RDF data, for which relational databases and specialized schemas can also be used. Hence, on the one hand it is desired to be able to process relational data as the RDF data, and on the other hand the RDF data, for the sake of efficiency, has to be stored in relational databases. This is because storage of the RDF data in a native form similarly to XML data faces problems with efficient indexing.

The main advantage of the relational representation is the efficiency of querying. This is in opposition to semantic data structures (Gomez-Perez, Corcho and Fernandez-Lopez, 2004), which usually are represented in the RDF format (Klyne and Carroll, 2004). Use of RDF based representations results in a lower query efficiency. To circumvent this problem Chong et al. (2005) implemented an extension to a RDBMS, based on a table function infrastructure, which allows for rewriting table functions with an SQL query. This extension introduces an RDF_MATCH table function, and 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 the RDF form.

Melton (2006) criticized using RDF and SPARQL as a query language to RDF structured data, claiming that efficiency of such a procedure is low, and the same effect can be achieved using SQL. Assuming required data availability in both data models, it is possible to propose an SQL query that is semantically equivalent to an SPARQL query (or more general - a graph based query). However, there are situations when SQL queries are much more awkward, harder to construct and less legible. So, it is tempting to keep asking queries based on graphs and get all the benefits of RDF representation, but answer these queries using the SQL engine. This approach does not require a permanent data transformation from the relational form into the RDF form and can be very efficient and scalable. Moreover, it is easy to implement this approach to work with existing systems, as no legacy system changes are needed. These advantages have been noticed and there is an increased activity in the area of incorporating relational data into the RDF-based reasoning systems. During the W3C workshop (W3C, 2007) on an RDF access to relational databases, a number of proposals and systems have been presented that serve this purpose. For example, the Virtuoso system (Erling and Mikhailov, 2007) uses database views and some rewriting algorithms to represent and query relational data in the RDF form, and the D2RQ (Bizer et al., 2006) is an application that transforms relational data to RDF data on demand and transparently to a client application. Recently, we presented a method that transforms a query to an RDF data structure (i.e. SPARQL, Prud'hommeaux and Seaborne, 2007) to an SQL query executed by an RDBMS (Falkowski and Jędrzejek, 2007). In this work that method is extended to handle one-to-many and many-to-many relationships. It will be also demonstrated that our method of enabling an RDF access to the relational data that relies on a query rewriting has some advantages over other approaches, as discussed in Section 2.

The paper is organized as follows. Section 2 discusses the need of expressing data semantics and reviews relevant data structures, mainly RDF. It is also shown in Section 2 how relational data can be transformed into RDF data. Section 3 discusses a graph matching problem. In Section 4 we present the architecture of a reasoning system that uses SQL queries to extract semantic data from relational databases. In Section 5 an algorithm of transforming a graph query to an SQL query, based on a mapping between predicate labels and relational table columns, is shown. Section 6 gives conclusions and future work prospects.

2. Semantic data structures

In the world of today, with many connected computer systems that share information, the problem of common data understanding is crucial. Applications often operate on data from many different sources and have to deal with differ-

ent naming conventions, different languages, metrics and others factors. Lack of precise data meaning can lead to serious problems, but data structures and models that are currently most popular (that is, the relational data model and databases for machine processable data, and HTML for human readable data) do not provide facilities to express data semantics. In the Semantic Web project a new data model was developed, which together with domain conceptualizations (called ontologies) allows to express data and its semantics in an easy, shareable and reusable way. This data model, called RDF (Resource Description Framework) is at the time the most common method of expressing data semantics (Klyne and Carroll, 2004). The base element of the RDF model is a triple: a resource (the subject) is linked to another resource or literal (the object) through an arc labeled with a third resource (the predicate). The meaning of such a triple is that <subject> has a property <predicate> valued by <object>. For example, the triple in Fig. 1 could be read as "John Black Has Telephone No. 1234567898".

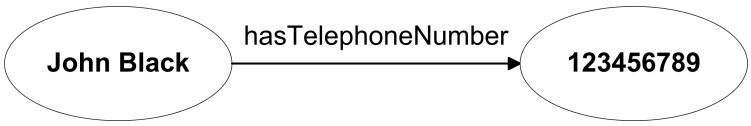

Figure 1. An example of the RDF triple

According to the RDF standard (Klyne and Carroll, 2004), subject and predicate labels have to be URI identifiers, and object can be a URI or a literal value. However, in this paper, for the sake of brevity, we name all RDF labels with short terms not in a URI form. The RDF alone is just a data model, and its ability to express semantics lies in that URI labels can be concepts (classes or properties) from a common ontology. In this context ontologies can be regarded as dictionaries, and have the necessary features that allow them to precisely define any relation or class and to describe them in different languages, and it is always possible and easy (because of URI naming) to check what a particular RDF triple means. Based on the RDF a few languages were developed, allowing for building ontologies. First was the RDF Schema (RDFS) (Brickley and Guha, 2004), which introduced classes and some properties, such as: rdfs:Class, rdfs:subClassOf, rdfs:subPropertyOf, rdfs:range, rdfs:domain, and others. It allows for building a simple concept hierarchy. OWL (Patel-Schneider, Hayes and Horrocks, 2004) is built on RDF and RDF Schema and adds more properties and expressive power; in particular, relations between classes (e.g. disjointness), some applications of cardinality (e.g. "exactly one"), the equality, a richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes. OWL has three increasingly-expressive sublanguages: OWL Lite, OWL DL, and OWL Full (however, OWL Full is not decidable). OWL DL is a subset of OWL Full that can assure existence of a decidable reasoning procedure. The

RDF data model is simple but powerful and it can be used to express any kind of information. It is also possible to transform data between the relational data model and the RDF data model. To do such a transformation it is necessary to know the source (relational) data semantics. In the simplest case a relational table represents an entity set and each row represents an instantiation of entity properties (values of table columns). Table name can be viewed as an entity class name and based on that and the table primary key value, an RDF identifier (URI) can be built. Names of columns are predicates and values of columns in a tuple are objects. An example is represented in Table 1.

Table 1. Table *Person*

Person					
ID	First name	Last name	IdCardNo	Born	Place of living
1	Matthew	Black	81070102357	01.07.1981	Warsaw

The entity *Person* can be represented in the form shown in Fig. 2.

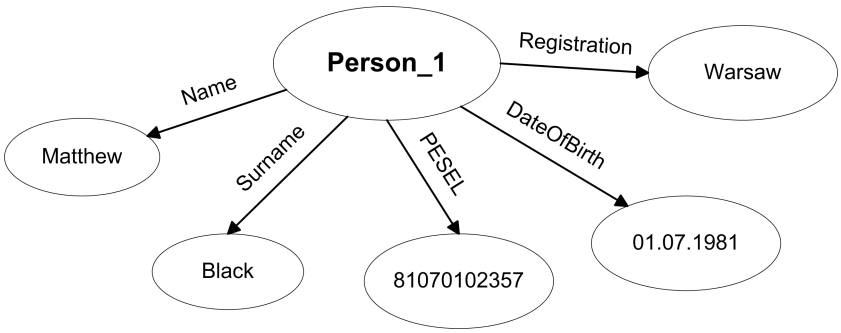

Figure 2. Entity *Person* in RDF representation

In this example we assume that ID is the primary key of a table. In such a case every row of the table contains information about an entity named Person_?x, where ?x is the row ID value. Relations between entities (tables) are expressed in the relational data model by a foreign key constraint. The value of a foreign key constrained column is not a literal value (such as a name or a date of birth), but another entity, identified by its unique ID (for example, father's ID). In the relational case this foreign ID (a foreign key) points to the target table and its primary key. In the RDF case, upon knowing the name of the target table and the value of its primary key, RDF identifier can be built, as mentioned earlier. Table 2 and Fig. 3 contain an example of such a relational schema and the corresponding RDF data.

Table 2. Table *Person* and table *Doctor*

Person			
ID	First name	...	DoctorID
1	Matthew	...	1

Doctor			
ID	First name	Last name	...
1	Steven	Cure	...

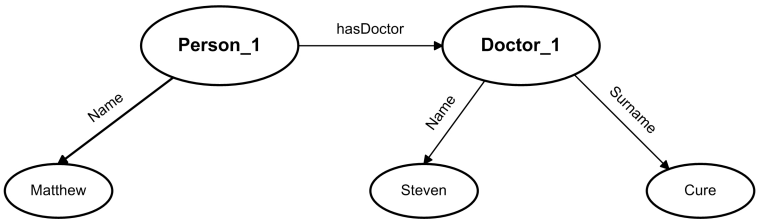

Figure 3. Entities Person, Doctor and their relation in RDF representation

Using a foreign key in the way described above can be helpful in expressing one-to-one or one-to-many relationships. Modeling many-to-many relationships between tables requires a separate table, consisting of foreign keys of relationship entities (e.g. patient's ID, doctor's ID and diagnosis ID) and optional relationship properties (e.g. date). In the simplest case, when the relationships table consists of two foreign key columns only (Table *Visit* in Table 3), it can be modeled as a single RDF property, named after that table name. If there exist more than two such columns (Table *VisitWithDate* in Table 3), for example an additional relationship attribute (such as date) column, it is necessary to define a separate entity. This is because all RDF statements are binary and it is not possible to express relationship attributes directly. Examples of such cases are shown in Figs. 4, 5 and Table 3.

Table 3. Table *Visit* and table *VisitWithDate*

Visit	
PatientID	DoctorID
1	1
1	2
2	1

VisitWithDate		
PatientID	DoctorID	Date
1	1	01.01.2007
1	2	01.02.2007
2	1	01.03.2007

In the remaining figures in this section relations marked with solid lines come from *Visit* (and *VisitWithDate*) tables directly, and relations marked with dotted lines come from tables associated with them through foreign key constraint and presented earlier (table *Person* and table *Doctor*).

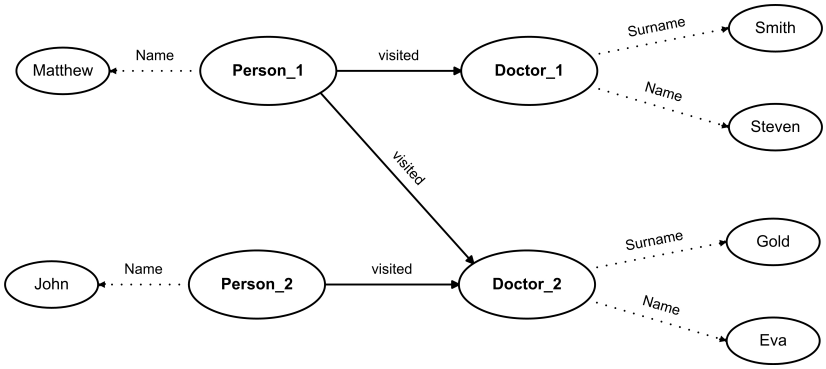

Figure 4. Entities of the *Visit* table in RDF representation

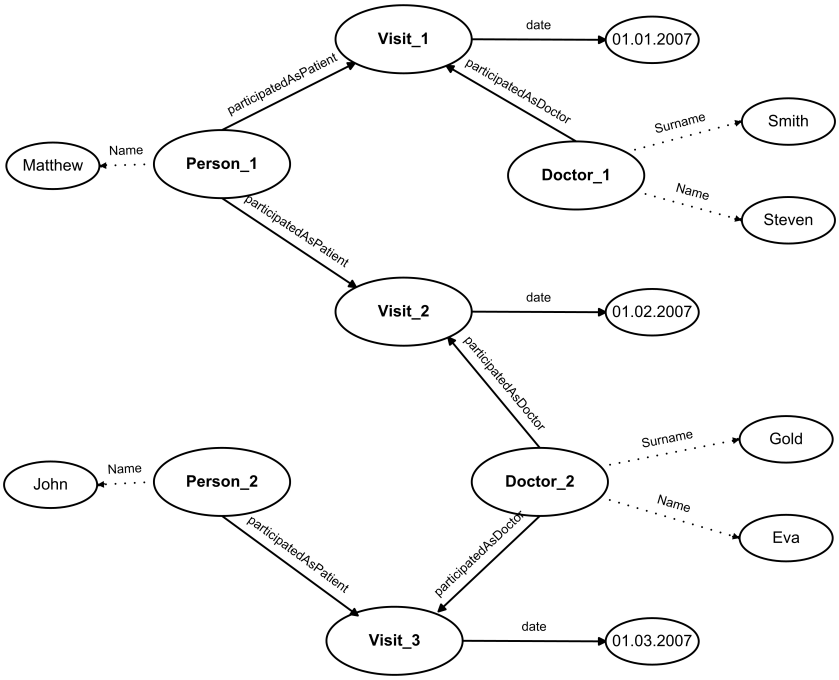

Figure 5. Entities of the *VisitWithDate* table in RDF representation

RDF representations of similar relational tables differ. Two-column table *Visit*, containing foreign keys only, can be modeled as a single RDF property ('visited') between entities referenced by foreign keys, as shown in Fig. 4. For the sake of clarity we omitted the second possible property, the reverse to 'visited',

from the *Doctor* entity to the *Patient* entity. Modeling a table as a property is not possible for the second table, *VisitWithDate*. That table contains an additional parameter, namely Date, and because of that has to be modeled as a separate entity, named *Visit* (Fig. 5).

3. Basic RDF graph definitions

Our formalism is similar to the one used in Flesca, Furfaro and Greco (2006) but specialized to RDF graphs.

3.1. Data graph, query graph, graph matching and a 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:

$$t = (s, p, o) | s, p \in \Gamma_U, o \in \Gamma_N. \tag{1}$$

The RDF triple consists of three elements: a subject (s), a predicate (p) and an object (o), of which the first two (s, p) cannot be labeled with literals. Elements s and o are vertices of the graph, and p is an arc between them. The components of a triple t will be denoted by $S(t)$, $P(t)$ and $O(t)$ respectively for a subject, a predicate and an object of the triple. Note that in the RDF model it is not allowed to have graphs with nodes not connected to any other node, and that a basic element of the graph is a triple; neither 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 the graph X.

Graphs differ according to allowed label sets used to constitute graph triples. A graph made of triples where $\Gamma_U = \Gamma_{URI}$ (and thus $\Gamma_N = \Gamma_{URI} \cup \Gamma_L$) is called a 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 matches the pattern defined by Q. For this, we define a function which unifies a query graph with a given data graph, and we define a new entity, called a mapping pair, consisting of a query graph and a mapping (unifying) function, which associates query graph variable nodes with data graph nodes.

3.2.1. Definition 1 - label 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 $\Gamma = \Gamma_{URI} \cup \Gamma_V \cup \Gamma_L$. A triple t matches the 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 the 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 a graph D consists of a query graph Q a and mapping function ζ, $M_D = (Q, \zeta)$. If a mapping pair exists for a given data graph and a query graph, then the answer for this 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 an element of the data; constant elements of the query are mapped to exactly the same elements in the data graph, and thus the most interesting part of mapping is the variable mapping - every variable is mapped to a value, and that mappings can differ between answers. On the basis of a mapping pair M_D, it is easy to create an answer graph Q^A - the graph of the same structure as the query graph Q, but with variable labels replaced by a function ζ, in such 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. 6.

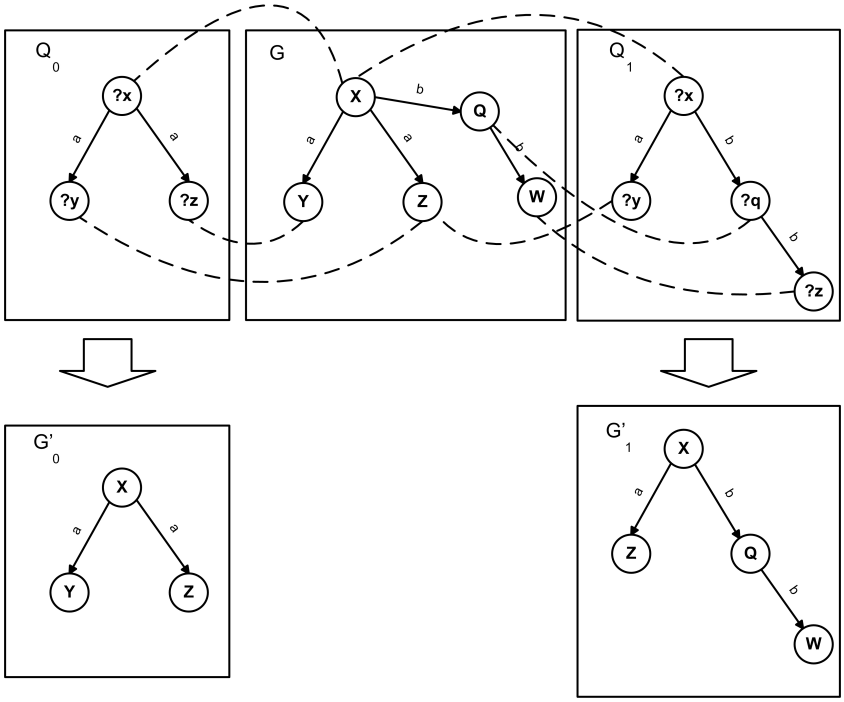

Figure 6. An example of matching graphs.

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 the head of a rule r, and $B(r)$ denotes the 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, in which variables can appear, and are matched if a matching function exists, such that triple_match(ζ, $H(r)$, t), (9), is true.

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.

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

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

The set $T_n \in t_q$ consists of triples for which at least one rule from the rule set can be matched. $T_t = t_q \setminus T_n$ 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 (a 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 for the matched triple, and of triples of the body of a rule with labels substituted 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 the parent node. An example of such a derivation is shown in Fig. 7. The only rule used here has the following form:

(?x a ?y) :- (?x b ?q) (?q b ?y).

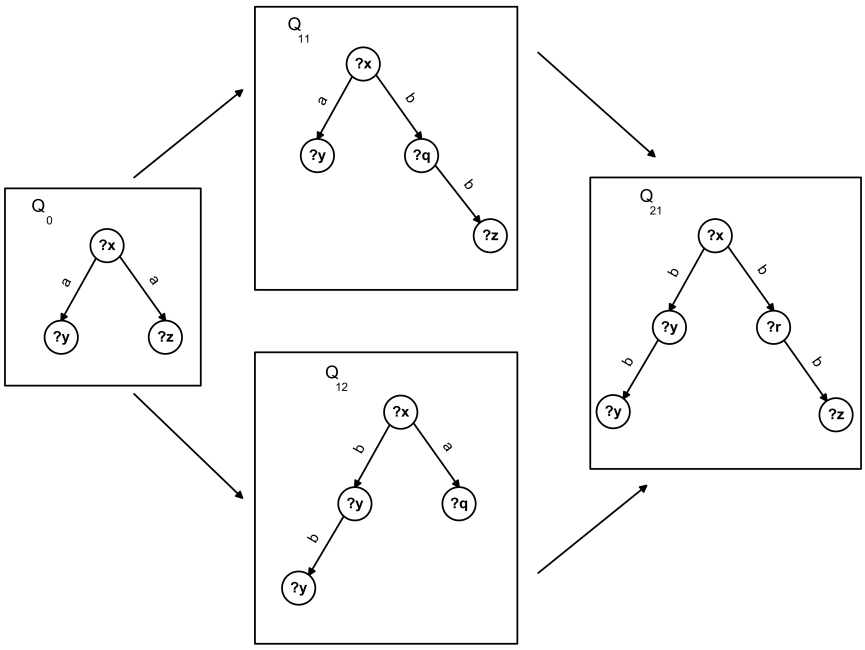

Figure 7. An example of a query graph processing with a rule

3.4. The use of the formalism

The 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), whose nodes 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 existing ones. Obtained rules consist of a 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 a twofold use of a predicate hasFather as seen in Fig. 8.

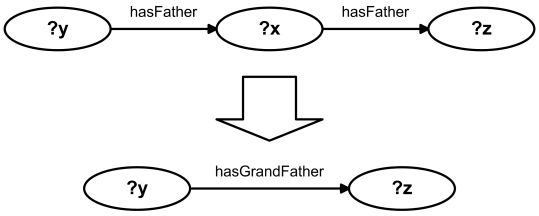

Figure 8. An example of a graph transforming rule

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

In the current data management environment most structured data reside in relational databases. Semantic data have to be extracted from text (HTML) or made from relational data. As shown before, a transformation between the relational data model and the RDF data model is possible. There are three main methods of utilizing relational data in RDF systems. Conceptually, all of them rely on relations between relational data tables and columns, and RDF predicates and classes (Berners-Lee, 1998). The first and conceptually simplest approach is a one-time permanent transformation. Based on relational data and transformation rules, RDF data are generated. In such a process an RDF copy of the relational data is created. The problem here is to store and efficiently query those data, and special schemas of relational tables can be used. An example of such a storage facility is the Oracle's RDF_TRIPLE data type, which is designed to store RDF triples in relational tables. It is also possible to develop specialized schemas, based on RDFS or OWL concepts, or even ontology classes and properties that can store semantic data. These approaches are less generic, but can be better suited for specialized systems. The general advantage of a permanent data transformation is the possible increase of query efficiency, as the transformation part is done once and in further operations (querying) is omitted. There are also disadvantages, such as maintaining and synchronizing both copies of data, the SQL one (for legacy systems) and the RDF one for semantic queries, which for large data volumes could be very expensive.

The second approach is to develop an on-demand data transformation adapter between an RDF application and a relational database. Such an adapter could use an appropriate mapping and generate RDF data from relational data "on the fly", transparently for the requesting application. An example of such a system is D2RQ (Bizer et al., 2006). The advantages of creating the RDF data on demand are: avoiding of data copy synchronization problems, always up-to-date RDF data and easy integration with existing database systems and RDF tools, because no database modifications are required. To enable RDF access to relational data with such an adapter it is sufficient to define how the RDF data is to be created from the relational data, and when an RDF application requests

such a data, the adapter formulates and executes SQL query, transforms results to the RDF form and passes it to the application. The main drawback is the low efficiency. This is because the adapter only extracts data, and it has to be processed at the application side. At the moment the data leave database all the profits of indexing and efficient data storage are lost. The RDF data querying is done at the client side and does not exploit mature and fast data retrieval algorithms implemented in the RDBMS. The example of a system architecture is presented in Fig. 9.

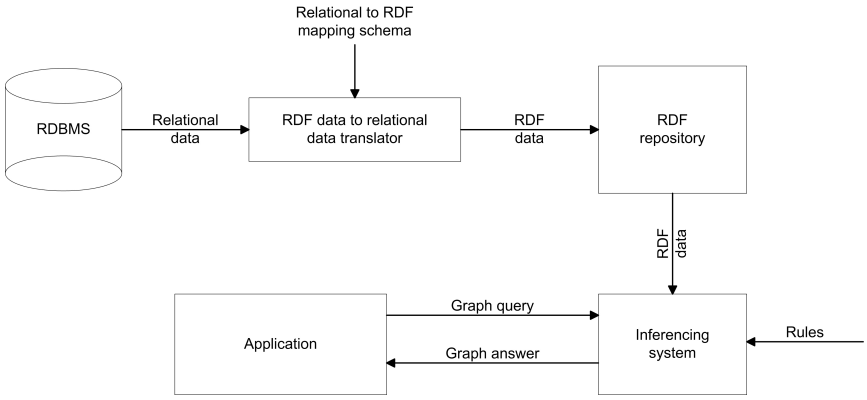

Figure 9. A traditional reasoning system architecture

The central part is an inferencing subsystem such as Racer Pro, Pellet, or Jena2. Data comes from a repository usually populated by ontologies, native RDF data (not shown in Fig. 9), and virtual RDF data created by a subsystem such as D2RQ. This data is queried by an application, possibly with the use of a set of rules or ontologies. The inferencing system provides an answer to the controlling application through a complex flow of data. The relational data have to be (possibly virtually) transformed to an RDF format. After this step the semantics of information does not change, but many advantages of the RDBMS efficient data storage are lost (such as B-trees and indices). As a result, the time needed to answer queries may be significantly longer than for the equivalent queries to relational data.

The third approach is to transform queries instead of data and push as much processing (data extraction mainly) into the RDBMS as possible. The results, in the form of relational tuples, can then be transformed to RDF triples and passed to a semantic, RDF application. The advantages of such a method, that is scalability and efficiency, come off data processing in RDBMS. Another advantage is a seamless integration with existing database systems, like in the second approach. The drawback is that client applications are restricted to use the backward reasoning only. The forward reasoning cannot be done because

of lack of data instantiations. An example of a complete RDF system architecture that uses backward inference engine and a query transformation module is shown in Fig. 10. Preliminary tests of such a system proved a good efficiency in comparison to existing applications.

The proposed reasoning system architecture is shown in Fig. 10. Functions

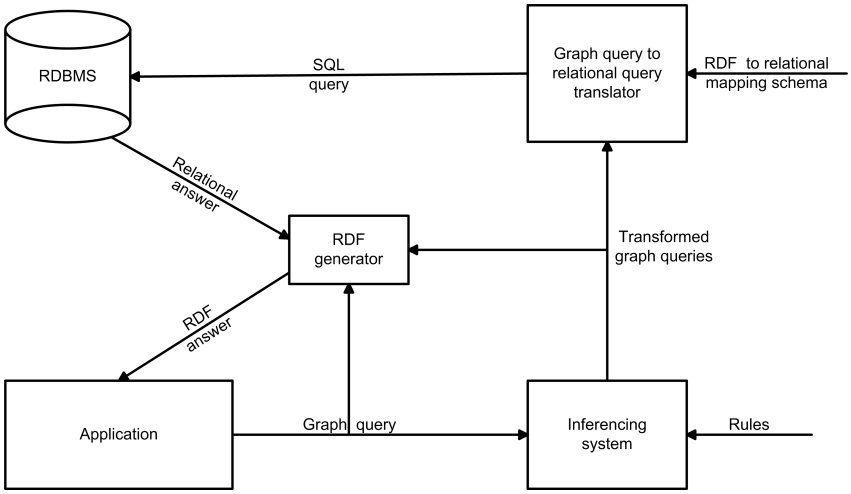

Figure 10. An architecture of reasoning proposed in this work

of the reasoning system have been divided into two 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. The second subsystem translates graph queries to elementary 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 to the RDF data and returned to the controlling application. The reasoning subsystem is optional, and if a query graph is a terminal graph, it can be omitted. Table 4 presents the summary of the three presented methods.

5. Query graph to SQL query translation algorithm

In this section a query rewriting algorithm is presented. For the sake of simplicity we assume that only one-to-one and one-to-many relations between tables are present in the relational schema over which the query is asked. Including many-to-many relations requires some small changes and extensions to the presented method, but would affect clarity and were omitted.

Table 4. Summary of methods of utilizing relational data in RDF systems

Method	Advantages	Disadvantages	Description
Permanent data transformation	Possible high efficiency due to specialized storage	The need of maintaining and synchronizing data copies	Takes data in relational form and outputs in RDF form
Online data transformation	Seamless integration with existing databases, always up-to-date data	Low efficiency	Takes data in relational form and outputs in RDF form on demand
Query rewriting	Always up-to-date data, seamless integration with existing databases, good efficiency (Falkowski and Jędrzejek, 2008)	Client applications can use backward reasoning only	Takes SPARQL query and outputs SQL query

5.1. The algorithm

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

To translate a query graph (a query to RDF data model) to an SQL query (a query to relational schema) it is necessary to know a mapping between RDF predicate labels and corresponding relational tables and columns. This mapping can be done by a function $m : \Gamma_{URI} \rightarrow$ tab.col. Function m maps URI labels to 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 a given group has the same subject:

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

2. Associate a relational table $T(g_i)$ with every group g_i, such that predicates of all triples within a 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 a unique table alias a_i to every group g_i.
4. For every triple $t \in Q$:
   (a) Assign a mapping string W to every subject that is a variable: S(t) $\in \Gamma_V \Rightarrow W(S(t)) = a_i.ID$
   (b) and a 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
   (c) Assign a mapping string E to every object that is a variable: O(t) $\in \Gamma_V \Rightarrow E(O(t)) = a_i.T(g_i)$
   (d) and a mapping string E' to every object that is not a 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 its alias
5. Add every element of W and E to SELECT part of a query.
6. Add every group alias a_i to FROM part of a query.
7. For every triple t $\mid S(t) \notin \Gamma_V$ add WHERE element of form: $W'(S(t)) = S(t)$
8. For every triple $t \mid O(t) \notin \Gamma_V$ 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 the form: $W(S(t)) = E(O(t'))$
10. The final result is a composition of SELECT, FROM and WHERE blocks.

5.2. Example of functioning of the algorithm

For illustration we demonstrate a query to the schema described in Table 5.

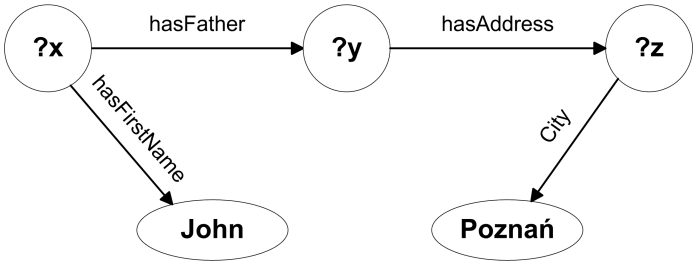

Figure 11. Example of a query

In the natural language the query from Fig. 11 would read: "Find persons whose first name is John and whose father has an address in the city of Poznan". In a relational model the relevant information can be stored in two tables: *Person* and *Address* (Table 5).

Table 5. Table *Person* and *Address*

Person				
ID	FirstName	LastName	Father'sId	AddressId

Address			
ID	PostalCode	Street	City

Mappings of properties into column names has the following form:

m = {hasFather: Person(Father'sId), hasFirstName: Person(FirstName), hasLastName: Person(LastName), hasAddress: Person(AddressId), Street: Address(Street), City: Address(City) }

These are input data to the algorithm. In the following, steps of the algorithm are presented according to Section 5.1.

1. Grouping triples according to their subject.
   The first step of the algorithm is to divide query triples into groups with the same subject. Each group represents one entity, and thus one tuple in an appropriate table. Dividing the graph in Fig. 11 gives three groups illustrated in Fig. 12.

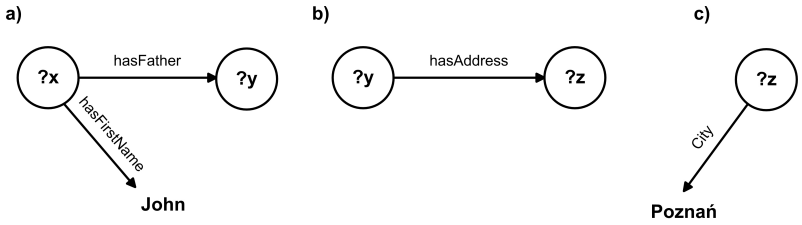

Figure 12. Query groups for a query in Fig. 11

2. Assigning relational tables to groups.
   The second step is to assign an appropriate table to every group. For example, group 'a' has two predicates: hasFather and hasFirstName, which both, according to m, are mapped to Persons table, columns Father's Id and FirstName, respectively. Similar to group 'a', other groups have to be processed, resulting in the following:
   group a - Table Person, alias O1,
   group b - Table Person, alias O2,
   group c - Table Address, alias A1.

3. Assigning aliases.
   Assigning aliases to groups has been shown in the previous step.

4. Assigning SQL statements:
   (a) to subjects that are variables
      - group a: W(?x) = O1.ID
      - group b: W(?y) = O2.ID
      - group c: W(?z) = A1.ID

   (b) to subjects that are not variables
   There are no non-variable subjects in the example query

   (c) to objects that are variables
      - group a: E(?y) = O1.Father'sId
      - group b: E(?z) = O2.AddressId

   (d) to objects that are not variables
      - group a: E'('John') = O1.FirstName
      - group c: E'('Poznań') = A1.City

5. Construct the SELECT part of the query
   The SELECT part is constructed of elements of W:
   SELECT O1.ID, O2.ID, A1.ID
6. Construct the FROM part of the query
   The FROM part of the query is formed using tables and aliases associated with particular groups (each entity (group) has its table and alias):
   FROM Person O1, Person O2, Address A1
7. Constructing the WHERE part of the query from subject constraints
   In the example query all subjects are variables and thus there are no constraints on their value.
8. Constructing the WHERE part of the query from object constraints
   There are two object constraints (elements of E' set):
      - group a: O1.FirstName = 'John'
      - group c: A1.City = 'Poznań'
9. Constructing the WHERE part of the query from object-is-subject constraints
   There are two variables (the intersection of W and E sets) that function as an object and a subject:
      - group a: O2.ID = O1.Father'sId
      - group c: A1.ID = O2.AddressId

   Conditions from 7, 8 and 9 together form the WHERE clause:
   WHERE O1.FirstName = 'John' AND A1.City = 'Poznań' AND O2.ID = O1.Father'sId AND A1.ID = O2.AddressId
10. The final result
    Concatenated query parts constructed in 5, 6, 7, 8 and 9 form the final query:

SELECT O1.ID, O2.ID, A1.ID FROM Person O1, Person O2, Address A1 WHERE O1.FirstName = 'John' AND A1.City = 'Poznań' AND O2.ID = O1.Father'sId AND A1.ID = O2.AddressId

6. Summary

In this paper we presented a method that translates a query to a graph into an SQL query to relational data of the same semantics as the original (virtual) RDF data. In the process of this translation, the use of rules (or ontologies) is allowed to transform original query to terminal queries. This method divides reasoning with rules into two steps - expanding queries according to rules (with backward reasoning) and answering these queries. The last step exploits the 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. In many cases a graph-model query provides better readability than a direct SQL query, and our method allows for asking readable and easy to formulate graph queries. In a future work we will concentrate on extending graph queries with additional constructs, such as cardinality constraints on the number of occurrences of some entity in a given relation, which we find useful and possible to translate to SQL queries. We will also try to employ an external rule reasoner, such as Prolog, to transform graph queries efficiently. One of main obstacles to the widespread use of semantic data is low efficiency of queries to semantic structures (possibly several hundred times compared to equivalent SQL queries). In this paper we were not concerned with efficiency of our method, but we have demonstrated elsewhere (Falkowski and Jędrzejek, 2008) that much progress in this area can be achieved.

Acknowledgment

This work has been supported by the Polish Ministry of Science and Higher Education, Polish Technological Security grant 0014/R/2/T00/06/02 and by 45-083/08/DS grant.

References

Berners-Lee, T. (1998) Relational Databases on the Semantic Web. *W3C Design Issues*. http://www.w3.org/DesignIssues/RDB-RDF.html.

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.

Brickley, D., and Guha, R. (2004) RDF Vocabulary Description Language 1.0: RDF Schema. *W3C Recommendation* REC-rdf-schema-20040210,

February 2004.

Chong, E., Das, S., Eadon, G. and Srinivasan, J. (2005) An efficient SQL-based RDF querying schema. *Proceedings of the 31st International Conference on Very Large Data Bases, VLDB'05*, Tronheim, Norway. VLDB Endowment, 1216–1227.

Erling, O. and Mikhailov, I. (2007) RDF Support in the Vurtuoso DBMS. *Proc. of the 1st Conference on Social Semantic Web. Lecture Notes in Informatics* **P-113**. Bonner Köller Verlag, 59–68.

Falkowski, M. and Jędrzejek, C. (2007) An efficient SQL-based querying method to RDF schemata. *II Krajowa Konferencja Naukowa - Technologie Przetwarzania Danych.* Wydawnictwo Politechniki Poznańskiej, 162-173.

Falkowski, M. and Jędrzejek, C. (2008) Efficiency of SQL-based querying method to RDF schemata. *Studia z automatyki i robotyki.* Wydawnictwo Politechniki Poznańskiej.

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.

Gomez-Perez, A., Corcho, O. and Fernandez-Lopez, M. (2004) *Ontological Engineering: with Examples from the Areas of Knowledge Management, e-Commerce and the Semantic Web.* Springer Verlag.

Heimbigner, D. and McLeod, D. (1985) A federated architecture for information management. *ACM Transactions on Information Systems (TOIS)*, **3** (3), 253-278.

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"*, http://xtech06.usefuline.com/schedule/paper/119.

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.

W3C Workshop (2007) *Workshop on RDF Access to Relational Databases*, Cambridge. http://www.w3.org/2007/03/RdfRDB/