2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems

Towards Adaptive and Semantic Database Model for RDF Data Stores

Svatopluk Šperka, Pavel Smrž
Brno University of Technology
Faculty of Information Technology
IT4Innovations Centre of Excellence
Božetěchova 2, 612 66 Brno, Czech Republic
Email: {isperka, smrz}@fit.vutbr.cz

***Abstract*—RDF Schema is a basic and yet very important language for specifying ontologies in the context of Semantic Web. Ontologies can be used to obtain more information from that explicitly stated. Traditionally, the process of revealing implicit knowledge, known as inference or reasoning, is realised by a reasoner – a component which either processes data to infer all conclusions in advance or, given a query, infers all implicit answers during a query evaluation process. In this paper, we present an alternative approach based on structuring data in a database according to underlying ontologies. This knowledge structure is employed for directing query evaluation to data relevant for the given query. Thus, this method reduces reasoning to retrieving. We show that the proposed schema preserves the semantics of RDF and RDFS in standard use cases.**

***Keywords*-database model; triple store; RDF; RDFS; inference; semantics; Semantic Web**

I. Introduction

Ontologies allow capturing knowledge structures by identifying concepts and their relations within a domain. They can be used to obtain implicit information from that explicitly stated. Some systems materialize all facts inferable from given information by a forward chaining strategy. However, when applied to extensive knowledge bases and rich ontologies, this approach can easily exceed storage limits. Moreover, it is an expensive strategy when new statements are continuously added and existing ones are retracted or modified. This paper deals with the cases when it is infeasible to precompute implicit statements in advance, i.e. where they need to be inferred during an evaluation of the query.

We focus on the problem of the in-query-time reasoning with ontologies specified in *RDF Schema*. It is a basic language for expressing ontologies in the context of *Semantic Web*. It is based on and intended for its standard data model – *Resource Description Framework* (*RDF*). RDFS allows specifying hierarchies of classes and relationships and declaring types of individuals participating in specific relations. Although RDFS is lean compared to *Web Ontology Language* (*OWL*) that offers more expressive constructs, it is still extremely useful for basic domain modelling.

Traditionally, reasoning is carried out by a specialised component which has to frequently interact with the data. No matter whether the reasoner is tightly or loosely coupled with the data store, interactions between storage itself and a reasoner are computationally expensive.

This paper proposes an alternative approach to RDF Schema reasoning. It does not require any kind of symbolic inference and thus effectively dissolves a reasoner as a specialised component.

The principle of the proposed technique lies in modelling of ontological concepts and relations directly in the database schema. A separate table is created for each concept and relation defined in an ontology. Statements using *ontology-structuring properties* from RDFS vocabulary are reflected into references among tables. These references are then used for directing the evaluation of queries to tables which are potential sources of consequences of deduction rules if reasoning were of a symbolic nature.

The following section introduces basic concepts of RDF and SPARQL query language that are used in the rest of the paper. Section III presents the model in detail. The query evaluation strategy is discussed in Section IV. Then, we show that the introduced approach preserves the semantics of RDF and RDFS. Related research is briefly outlined in Section VI. The paper concludes by discussing directions of our future work.

II. Preliminaries

To present the proposed database model, let us first introduce key concepts related to the RDF data model and the SPARQL query language that is used to query RDF data.

The set of RDF terms ($RDF\text{-}T$) is the union of three pairwise disjoint sets – I, $RDF\text{-}L$ and $RDF\text{-}B$. I is the set of *Internationalised Resource Identifiers* or *IRI*s [1]. They are used to identify resources, both abstract and physical. $RDF\text{-}L$ is a set of literals – direct values such as strings, numbers or dates. The last set, $RDF\text{-}B$, consists of blank nodes which are used to refer to unidentified (anonymous) resources.

The basic unit of expression in RDF is a *triple*. It is an atomic statement of the form *subject-predicate-object*. A triple can mathematically be represented as a tuple $(s, p, o) \in (I \cup RDF\text{-}B) \times I \times RDF\text{-}T$. If both subject and predicate are IRIs, the triple states a specific relationship,

978-0-7695-4687-2/12 $26.00 © 2012 IEEE
DOI 10.1109/CISIS.2012.137

given by the predicate element, between two entities. The triple is to be interpreted as stating a value of an attribute of a resource identified by the subject in the case that a literal takes the position of the object. Blank nodes are interpreted as existential variables in RDF triples. The definition of RDF triple poses syntactic constraints on what can appear as the subject and the predicate. Subjects can only be IRIs or blank nodes. This means that one can only talk about resources and not about direct values. There is also a restriction on the predicate – an identity of a relationship has to be known and it has to be an IRI. A set of RDF triples is called an *RDF graph* or a *dataset*.

SPARQL is the primary language for querying RDF data [2]. The basic building block of a SPARQL query is a *triple pattern*. It serves as a template for triples in a dataset. For all triples that match the pattern, values on positions of variables (prefixed by `?`) in the query are bound to these variables in a solution. Variables come from a set V disjoint from $RDF\text{-}T$. A triple pattern can then be defined as $p \in (I \cup RDF\text{-}B \cup V) \times (I \cup V) \times (RDF\text{-}T \cup V)$. A solution of a SPARQL query is defined in terms of a multiset Ω of partial functions (i.e. a variable may be unbound in a solution) $\mu : V \to RDF\text{-}T$ called *solution mappings*.

A query is transformed into the SPARQL *algebra* [2, section 7.4] in order to be evaluated. The algebra provides three operators for forming more complex *graph patterns* (a triple pattern is the *basic graph pattern*; *BGP*) – *Join* ($\bowtie_S$[1]), *LeftJoin* ($\bowtie_S$, OPTIONAL in SPARQL syntax) and *Union* ($\cup_S$, UNION in syntax). *Join* is the implicit operator and it combines two solutions Ω_1 and Ω_2 such that $\Omega_1 \bowtie_S \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible}\}$. Solution mappings μ_1, μ_2 are compatible if, for every variable in a domain of μ_1 and of μ_2, $\mu_1(v) = \mu_2(v)$. *Union* of two mappings is defined as $\Omega_1 \cup_S \Omega_2 = \{\mu \mid \mu \in \Omega_1 \vee \mu \in \Omega_2\}$. For *LeftJoin* we need a difference between two solutions – $\Omega_1 \setminus_S \Omega_2 = \{\mu \in \Omega_1 \mid \forall \mu' \in \Omega_2 : \mu \text{ and } \mu' \text{ are not compatible}\}$. *LeftJoin* itself is then $\Omega_1 \mathbin{⟕}_S \Omega_2 = (\Omega_1 \bowtie_S \Omega_2) \cup (\Omega_1 \setminus_S \Omega_2)$ [2][3].

III. Database Model

The fundamental idea is to reflect an RDFS ontology (or ontologies; there is no difference in principle) into a database schema. We translate terminological knowledge into a set of tables while maintaining relations between these tables in the form of references, i.e. in a non-symbolic way. The model is thus inherently adaptive as the schema is specific for the particular set of ontologies. Relationships between ontological entities are employed when the database is queried in order to provide implicit triples.

The translation to a specific schema involves *ontology-structuring properties* only. These properties come from the RDFS vocabulary and they have two important features. Firstly, they strictly relate terminological entities and, secondly, they take part in entailment rules defined for RDFS [4, section 7]. Ontology-structuring properties are `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain` and `rdfs:range`. The first couple allows specifying class and property hierarchies, respectively. The second pair allows defining types of resources connected by a property described using these constructs. Notice that `rdf:type` which assigns a type to a resource is not included because it may relate a terminological entity and an individual.

The database schema D is defined as $D = (C, P)$, where C is a finite set of relational tables of type $Class$ (class tables) and P is a finite set of tables of type $Property$ (property tables). Tables are created when triples of a certain form are encountered during a loading of an RDF dataset into the database. We forbid a non-standard use of RDFS vocabulary [5], particularly of ontology-structuring properties. This means that these properties cannot be used as subjects and objects of triples because that would lead to a redefinition of RDFS [6].

Tables of type $Class$ are of the form $iri(member)$, where $iri \in I$ is the name of a relation and $member \in I \cup RDF\text{-}B$ is the only attribute. Class table contains members of the class and thus rows of the table are given by loaded triples of the form `any:iri1 rdf:type any:iri2`, i. e. $iri1$ becomes a row in the table $iri2$.

Tables of type $Property$ have the structure $iri(subject, object)$ where $iri \in I$ is the name of a relation. It follows from the definition of the RDF triple that $subject \in I \cup RDF\text{-}B$ and $object \in RDF\text{-}T$. A property table emerges when a triple of the form `any:iri rdf:type rdf:Property` or `any:iri1 any:property any:iri2` is being imported (where `any:property` is not an ontology-structuring property or `rdf:type`, i. e. an IRI not interpreted by the system). A row $(iri1, iri3)$ is created in the table $iri2$ when the triple of the form `iri1 iri2 iri3` is encountered.

Table I
Metadata entries for class and property tables

`Class`	$subClasses \in \mathcal{P}(I)$	`Property`	$subProperties \in \mathcal{P}(I)$
	$superClasses \in \mathcal{P}(I)$		$superProperties \in \mathcal{P}(I)$
	$domainOf \in \mathcal{P}(I)$		$domains \in \mathcal{P}(I)$
	$rangeOf \in \mathcal{P}(I)$		$ranges \in \mathcal{P}(I)$

Tables are relational but they have additional metadata attached to them. The metadata can be thought of as attributes in object-oriented sense (if a table is considered to be an object). It contains an entry *iri* that determines the identity of a class or property table. Other entries are collections of references to related tables according to triples using ontology-structuring properties that were encountered during data loading. Table I shows these metadata entries

[1]We use S in subscript to denote operators of SPARQL algebra. Symbols without it will be used to denote operators of the relational algebra.

for both types of tables. They determine evaluation of triple patterns of SPARQL queries. In order to facilitate evaluation of patterns involving ontology-structuring properties, they capture these relationships in a bidirectional manner – only a part of them is needed for reasoning itself. For example, *domains* of property tables is never used for reasoning because properties are sources of members for domain classes. Thus, a class needs the direction to a property it is a domain of, not vice versa. Evaluation strategy is described in detail in Section IV.

Axioms for both RDF [4, section 3.1] and RDF Schema [4, section 4.1] need to be true in the database. This way we can ensure that the query evaluation mechanism described in the next section is correct. We ensure presence of these axioms by loading definitions of RDF and RDFS.

Concepts like `rdfs:Resource` (the class of all resources), `rdf:Property` (the class of all properties) and `rdfs:Class` (a superclass of all classes) are defined in the RDF and RDFS vocabularies. Tables identified by IRIs of these concepts in our database schema can be used as a catalogue of individuals and terminological entities. The query evaluation process takes use of this catalogue to obtain sets of classes or properties present in the system. Let us take triple pattern `test:iri1 ?p test:iri2` as an example. To get all tables that need to be searched for a presence of the tuple (`test:iri1`, `test:iri2`), `rdfs:Property` can be employed.

IV. Query Evaluation

We have described crucial operators of SPARQL algebra in Preliminaries. Tables in our database model are of a relational nature and therefore we use the transformation of SPARQL operators into relational algebra described in [7].

SPARQL algebra's *Join* is translated to *inner join* ($\bowtie$) of the relational algebra. It joins tuples from input relations that equal on their shared attributes. Shared attributes are those that are bound in both tuples. *LeftJoin* operator can be mapped to the *left outer join* ($\mathbin{⟕}$). It does not discard tuples from the left relation for which there is no matching in the right one. Therefore, the solution can have unbound variables. *Union* operator of SPARQL is mapped onto *outer union* of two relations, i.e. the set union of tuples in both relations. Attributes unbound in an input relation stay unbound in the resulting one.

Some basic operations of the relational algebra are also needed. A *projection* to a set of attributes is designated by π. For example, $\pi_{subject}\,table$ denotes extracting the subject column from a property table. Binding values of matched triples to variables in a solution mapping can be modelled as a combination of the *projection* and the *rename* (ϱ) operators. For the triple pattern `?p foaf:name "xxx"`, values are bound to the variable `?p` by projecting `foaf:name` on the attribute *subject* and subsequently renaming it to `?p`, i. e., $\rho_{?p \leftarrow subject}(\pi_{subject}\textit{foaf:name})$ or more succinctly $\pi_{?p \leftarrow subject}(\textit{foaf:name})$. Filtering tuples based on constants, i. e., when a component of a triple pattern is given (as well as evaluating SPARQL expression `FILTER` which we do not consider here) is analogous to *restriction* (σ) of the relational algebra. In the former case, the restriction is based on equivalence.

The process of finding all solutions for a triple pattern is defined iteratively. The starting point depends on the form of the triple pattern:

- If the predicate of a triple pattern is the `rdf:type` then the starting point is the class table identified by the object of the pattern or all of class tables in the case that the object is a variable.
- When the predicate is an ontology-structuring property, we start from a class/property table explicitly mentioned in the pattern, either as the subject or the object. If both are variables, we have to take all class or property tables, depending on which particular property is used.
- If the predicate is any other property then the process starts at the table representing this property.
- The last case is the predicate being a variable – then the the process starts at the set of all property tables.

The process extracts content from starting tables and then follows relevant references stored in metadata. In such a manner the process is repeated until there are no tables to process. Cycles in references among tables are detected in order to prevent infinite loops in the evaluation.

Let us show which references are considered as relevant. For reasoning in a class or a property hierarchies, references in *subClasses* and *subProperties* need to be followed, respectively. Entities lower in the hierarchy are lifted to a concept or a property of explicit interest (that would be a reasoner's task in a symbolic processing system). This lifting ultimately satisfies semantic requirements of `rdfs:subClassOf` and `rdfs:subPropertyOf`. The evaluation process additionally follows *domainOf* and *rangeOf* references in the case of class tables in order to satisfy semantics of `rdfs:domain` and `rdfs:range`, i.e. to extract subjects/objects of properties that a particular class table is domain/range of.

Querying *ontology-structuring properties* is realised by obtaining entries in the tables' metadata while traversing tables in the same manner as for uninterpreted properties. For example, a triple pattern requiring subclasses of a given class is evaluated by traversing the class hierarchy and collecting *iri* metadata entries of all classes the process encounters – including the starting class in order to satisfy the reflexivity of subclassing.

Figure 1 presents the pseudocode of the algorithm for the evaluation of triple patterns that do not use ontology-structuring properties. It is easily extensible to be able to collect contents of metadata for solutions of such patterns but the point is to show the principle of the evaluation on the minimal example. We use a queue as a data structure

for holding tables to be processed. The queue is initialised by pairs consisting of starting tables and related relational operators that extract required columns. At each iteration, a pair is dequeued from the queue and current solutions are united with the result of an application of the operator to that table. In the next step, relevant tables from metadata entries of the currently evaluated table are queued along with relational operators. It is analogous to the initialisation phase. In order to prevent an infinite loop in the evaluation, it is also ensured that the same combination of a table and a relational operator is not evaluated twice.

```
solutions ← starting (table, operator) tuples
while !queue.empty do
  (t, op) = queue.dequeue
  if op( t ) was not evaluated in the context of this triple
  pattern then
    solutions ← solutions ∪ op( t ) {collect all relevant
    tables from metadata}
    if t.isClass then {collect all relevant tables – class}
      A ← map( λx.(x, π_member), t.subClasses )
      B ← map( λx.(x, π_subject), t.domainOf )
      C ← map( λx.(x, π_object), t.rangeOf )
      queue.enqueueAll( A ∪ B ∪ C )
    else {property}
      queue.enqueueAll( (t.subProperties, op) )
    end if
  end if
end while
```

Figure 1. Pseudocode for evaluation of a triple pattern without ontology-structuring properties.

For example, consider the evaluation of the triple pattern `?p rdf:type test:Person` captured in figure 2. The starting point is the class table for persons because `rdf:type` is used as the predicate and `test:Person` is used at the object. In the first step, the queue only contains a pair $(\texttt{test:Person1}, \pi_{member})$. Starting table has no explicit members and so there are no solutions yet. Its subclasses, however, are queued along with properties which `test:Person1` is the domain or range of (`test:participatesOnCourse`). The queue then contains two pairs – $(\texttt{test:AssistantProfessor}, \pi_{member})$ and $(\texttt{test:participatesOnCourse}, \pi_{subject})$.

All members of `test:AssistantProfessor` are added to the result in the second step. There are no references to be queued. Next, subjects of `test:participatesOnCourse` are taken into account. There are no pairs contained in this property table and therefore the result is still unchanged. Its subproperties are added to the queue because, again, subjects in triples using this property are wanted. `test:Person2` – the subject of the only pair in the table `test:teaches` – is added to the result in the last step. There are no more links to follow. Solutions thus contains `test:Person1` and `test:Person2`.

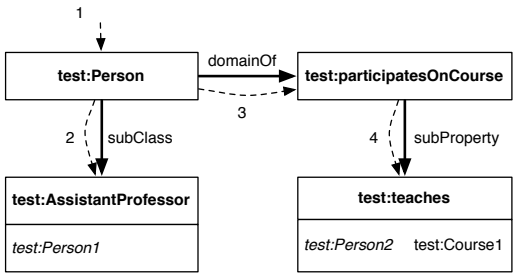

Figure 2. Visualisation of the evaluation of triple pattern `?p rdf:type test:Person`

Now when we are able to evaluate triple patterns we can combine their solutions according to the SPARQL algebra form of the query. That is, we use the bottom-up evaluation strategy of expressions [3]. For example the SPARQL pattern { `?s x:p1 ?v1 OPTIONAL {?s x:p2 ?v2 }` } is transformed into $\mathit{LeftJoin}(\mathit{BGP}(\texttt{?s x:p1 ?v1}), \mathit{BGP}(\texttt{?s x:p2 ?v2})$[2] [2]. Obtaining solution mappings proceeds by evaluating the left basic graph pattern (triple pattern), then the right one and followed by applying *LeftJoin* on their solutions.

V. Conformity with RDF and RDFS Semantics

Given the database model and the evaluation strategy described in previous sections, let us show that the proposed system preserves both RDF and RDFS semantics by implementing all relevant RDF/RDFS entailment rules.

As we noted in section III, we forbid usage of ontology structuring properties as subjects and objects of RDF triples. Such statements can lead to redefinition of RDFS [5] and it would cause difficulties in the the design of the query evaluation process. We also omit the datatype reasoning because it is not in the scope of this work.

It is necessary to note that the semantics is limited to what can be provided as solutions to queries according to the RDFS entailment regime of SPARQL 1.1 [8]. In particular, this means that no solution will contain blank nodes or IRIs of the form `rdf:_n` for any natural number n that is actually not present in the graph in consideration.

RDF and RDFS semantics is formed, as any logical theory, by axioms (in this case *axiomatic triples*) and *entailment rules*. These triples are contained in schemata for RDF and RDFS and, because we ensure their presence in the database, they are indeed always true.

Entailment rules of RDF and RDFS [4, section 7] are satisfied either by the query evaluation itself or by additional

[2]We omit filter conditions used in the source document.

steps during the loading of RDF data. We mention which method is used for each rule.

Rules of simple entailment (*se1*, *se2*, *lg* and *gl*) produce triples containing blank nodes only and these cannot be retrieved due to SPARQL's RDFS entailment regime. RDF entailment rule *rdf2* and RDFS rule *rdfs1* can be ignored for the same reason.

Rule *rdf1* (everything that occurs as a property in a triple is in fact of type `rdf:Property`) is satisfied by adding an IRI on the property position in a triple to `rdf:Property` table – but only if this property is not interpreted by the system (i.e. it is not an ontology-structuring property or `rdf:type`) and if it is not already contained.

To simplify the discussion in the rest of this section, let us employ the convention used in [4] for denoting elements of triples. In particular, *aaa* will be used for IRI; *uuu*, *vvv* for IRI or a blank node; *xxx* for IRI, a blank node or a literal.

Entailment rules *rdfs2* and *rdfs3* are analogous. They state that if a class *xxx* is a domain or a range of a property *aaa*, then subjects/objects of triples using this property are of the type *xxx*. These rules are satisfied by directing the evaluation of a query asking for members of *xxx* to subjects or objects of the table identified by *aaa*.

Rules *rdfs5* and *rdfs11* that capture transitivity of `rdfs:subPropertyOf` and `rdfs:subClassOf`, respectively, are satisfied because the evaluation of any query involving class/property *xxx* will also be directed towards *vvv* and subsequently to *uuu* for any *uuu* being a subclass/subproperty of *vvv* and *vvv* of *xxx*.

The reflexivity of `rdfs:subPropertyOf` and `rdfs:subClassOf` that is captured by rules *rdfs6* and *rdfs10* is satisfied trivially by adding the *iri* element from table's metadata to solutions of triple patterns asking for subclasses or subproperties of a given terminological entity.

Rule *rdfs7* simply states that if a pair of resources is connected by a specialisation of some property, it is also connected by this more abstract property. The analogous rule for classes is the rule *rdfs9*. This lifting of pairs/individuals in a property/class hierarchy is satisfied by following *subProperties*/*subClasses* references to tables stored in table's metadata and retrieving their contents.

For the rules *rdfs8*, *rdfs12* and *rdfs13*, it is necessary to add references into metadata of particular tables during the loading process. For *rdfs8*, a reference to a newly created class table needs to be added to *subClasses* of *rdf:Resource*. A property table must be created for every resource that is stated to be a `rdfs:ContainerMembershipProperty` (*rdfs12*) and a reference to it added to *subProperties* of `rdfs:member`. It is an analogous case for *rdfs13*.

Rules *rdfs4a* and *rdfs4b* represent a peculiar case because there is no premise in them involving ontology-structuring properties. We employ extensional semantics to satisfy them [4, section 7.3.1], i.e. we state that `rdfs:Resource` is a domain and a range of every property. When asking for everything of type `rdfs:Resource`, a content of any property table will be included.

VI. Related Work

If we use only property tables for all properties, including ontology-structuring properties and `rdf:type`, then we get a vertical paritioning schema for storing RDF introduced in [9]. The paper discusses usage of a column-oriented relational database in order to speed up query processing. A critique of this approach is linked to an evaluation of a triple patterns with a variable on the predicate position [10]. There is a danger of a large number of tables to be investigated if a lot of properties are used. That is usually the case for datasets with extensive domain scope (for example *DBpedia* contains 44469 distinct properties[3]). For more specifically oriented datasets, however, numbers are much lower. For example *SWDF*[4] uses 156 properties; *Jamendo* dataset hosted at dbtube.org that holds data from Jamendo music service uses 45 properties; *GenBank@NCBI* dataset which is part of the *Bio2RDF*[5] project uses 139 properties. Those numbers are not unusual for larger relational database schemas and can be handled by contemporary database systems.

The evaluation of the proposed model can be thought of as a deductive database with Datalog rules but without symbolic processing. Everything is encoded in the model itself and the way evaluation process works. With rules, the proposed query evaluation could be captured as shown in the following example:

$$person(X) \quad :- \quad person_e(X). \tag{1}$$
$$person(X) \quad :- \quad assistantProffesor(X). \tag{2}$$
$$assistantProffesor(X) \quad :- \quad assistantProffesor_e(X). \tag{3}$$

Predicates with e in the subscript are extensional (values for which they are true are stated explicitly). Predicates without it are intensional (values for which they are true can be deduced by symbolic reasoning). Ullman showed that Datalog queries can be translated into the relational algebra and thus implemented by relational databases [11]. Extensional predicates become tables in the relational database and intensional predicates become views. In the model proposed in the paper, references in metadata entries of a table can be seen as definitions of views. Triple store *DLDB* [12] and subsequent *DLDB2* [13] used this concept by employing materialised views. As discussed in the beginning, the materialisation of views is inappropriate for applications where data changes continuously. DLDB also implemented semantics of `rdfs:subClassOf` and `rdfs:subPropertyOf` only.

RDFSuite [14] follows a similar approach. Instead of views, it exploits the object-relational feature of SQL99

[3] As of 21st January 2012
[4] http://data.semanticweb.org
[5] http://www.bio2rdf.org

for representing subsumption relationships by using *sub-table* definitions. The semantics of the sub-table relationship states that contents of sub-tables are included in super-tables. Using this construct, the semantics of `rdfs:subClassOf` and `rdfs:subPropertyOf` is satisfied. On the other hand, reasoning with support for domains and ranges of properties cannot be specified this way. The biggest limitation of the sub-table approach is that SQL99 does not support multiple inheritance – a table cannot have more super-tables [15].

VII. Conclusion and Future Directions

This paper introduced a database model for storing information encoded in RDF with support for RDF Schema semantics. Unlike traditional approaches, the RDFS semantics is not realised by any form of a dedicated component employing symbolic reasoning. References among tables that reflect relationships of terminological entities specified in an ontology are used to direct the evaluation of a triple pattern to obtain all solutions – including those that are entailed by explicit information. This fact can improve performance as it excludes interactions between the reasoner and the stored data.

A prototype of the proposed model was implemented as a proof-of-concept demonstrating the conformance with the RDFS semantics. A new version is currently being developed with performance in mind. The realized solution will then be evaluated and its performance will be compared to that of other triple stores implementing the RDFS semantics. As the vertical partitioning method for storing RDF proposed in [9] is similar to our model, an open-source column-oriented relational database MonetDB will be used as a basis.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 256910 (mOSAIC Cloud) and by the IT4Innovations Centre of Excellence project, Registration number CZ.1.05/1.1.00/02.0070, supported by Operational Programme "Research and Development for Innovations" funded by Structural Funds of the European Union and the state budget of the Czech Republic.

References

[1] W3C Working Group, "RDF Primer (W3C Recommendation 10 February 2004)," 2004.

[2] W3C Working Group, "SPARQL 1.1 Query Language (W3C Working Draft 14 October 2010)," 2010.

[3] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of sparql," *ACM Trans. Database Syst.*, vol. 34, pp. 16:1–16:45, September 2009.

[4] W3C Working Group, "RDF Semantics," 2004.

[5] J. D. Bruijn and S. Heymans, "Logical Foundations of (e)RDF(S): Complexity and Reasoning," in *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007), Busan, South Korea* (K. Aberer, K.-S. Choi, N. Noy, D. Allemang, K.-I. Lee, L. J. B. Nixon, J. Golbeck, P. Mika, D. Maynard, G. Schreiber, and P. Cudré-Mauroux, eds.), vol. 4825 of *LNCS*, (Berlin, Heidelberg), pp. 85–98, Springer Verlag, November 2007.

[6] S. Muñoz, J. Pérez, and C. Gutierrez, "Simple and Efficient Minimal RDFS," *Web Semant.*, vol. 7, pp. 220–234, September 2009.

[7] R. Cyganiak, "A Relational Algebra for SPARQL," Technical Report HP-2005-170, Digital Media Systems Laboratory, HP Laboratories Bristol, 2005.

[8] W3C Working Group, "SPARQL 1.1 Entailment Regimes (W3C Working Draft 14 October 2010)," 2010.

[9] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach, "Scalable Semantic Web Data Management Using Vertical Partitioning," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, pp. 411–422, 2007.

[10] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold, "Column-store Support for RDF Data Management: Not All Swans are White," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1553–1563, 2008.

[11] J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[12] Z. Pan and J. Heflin, "DLDB: Extending Relational Databases to Support Semantic Web Queries," Technical Report LU-CSE-04-006, Dept. of Computer Science and Engineering, Lehigh University, 2004.

[13] Z. Pan, X. Zhang, and J. Heflin, "DLDB2: A Scalable Multi-perspective Semantic Web Repository," in *Web Intelligence and Intelligent Agent Technology*, pp. 489–495, 2008.

[14] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle, "The ICS-FORTH RDFSuite: Managing voluminous RDF description bases," in *2nd International Workshop on the Semantic Web*, 2001.

[15] C. Türker, "Schema Evolution in SQL-99 and Commercial (Object-)Relational DBMS," *Database Schema Evolution and Meta-Modeling*, pp. 1–32, 2006.