Experiences from a TBox Reasoning Application: Deriving a Relational Model by OWL Schema Analysis

Thomas Hornung[1] and Wolfgang May[2]

[1] Institut für Informatik, Universität Freiburg,
hornungt@informatik.uni-freiburg.de
[2] Institut für Informatik, Universität Göttingen,
may@informatik.uni-goettingen.de

Abstract. Given an OWL-DL ontology of an application domain, we derive an application-specific relational model as known from classical database design, where the relational model is created from an ER model. The mapping depends as usual on the cardinalities of properties; namely whether a property is functional or not on a certain domain.
In particular, since properties may be defined for several classes, where they can be functional on some of them, and multivalued on others, a detailed schema inspection is necessary. We show that this requires the definition of auxiliary classes at processing time followed by containment checks. Thus, the process cannot be implemented declaratively by a simple set of SPARQL queries, but requires the use of a procedural framework that interferes with the ontology.

1 Introduction

OWL Ontologies (sometimes) provide comprehensive knowledge about the concept level of an application: characterizations of the classes of relevant resources, the class hierarchy, properties, and relationships between them. Such information can be used, e.g., for designing a relational storage schema.

The relational model provides a well-established and well-supported technique for storing and accessing data. As a *fully-structured* data model, it is applicable for applications where the schema is known, and where the data is rather homogeneous, in contrast to semistructured data models like XML or RDF (as logical data models), or column stores (as a physical data model). SPARQL queries can then be translated into equivalent SQL queries over the relational storage, which is part of the future development of this work; additionally, the data is accessible by SQL.

Using traditional ER diagrams, the generation of a relational schema is rather simple: the classes are explicitly given with their attributes and relationships, and even only rarely, a simple class hierarchy is used. In contrast, an ontology is usually more complex, building on a richer class hierarchy. Thus, deriving the relational model cannot be done by only looking up attributes and relationships,

but incorporates reasoning about the ontology. We illustrate the paper by the MONDIAL case study[3]. The following examples give a rough idea of the difference between a plain ER model and the modeling in an ontology:

Example 1 (ER Model) *An ER model states that countries have functional attributes name, code, area, population, are in a one-to-one relationship* capital *and in a one-to-many relationship* hasCity *with one or more cities, and in an* $n : m$*-relationship* neighbor *with countries (the relationship has a functional attribute* length*), and an* $n : m$*-relationship* isMember *with organizations.*

Example 2 (Ontology) *The concept hierarchy in an ontology is usually much more detailed since also* abstract *classes are used: Most* Things *(except such things as borders, reified things, etc.) have a functional property* name*. There are* Areas *that have functional properties* area *and* population*, and that are in* $n : m$*-relationship* borders *with other areas. There are* AdministrativeAreas *that are areas, and have a (at most one) capital. Countries are both* Things *and* AdministrativeAreas*, they also have a code, are in a one-to-many relationship* hasCity *with one or more cities. The* neighbor *relationship is a subproperty of the* borders *relationship, with the range* Country*. The* borders *relationship between countries is reified into* Border*, which is a* Line*. Lines have a functional* length *property (that is then also inherited e.g. to rivers).*

Usage of Annotation Properties. In OWL-DL, classes and properties are not allowed to have "normal" properties. Information about abstract/non-abstract classes and about reification issues is represented by owl:AnnotationProperties in the er: namespace:

- c er:isa er:Class/er:Concrete/er:Abstract/er:Interface applies to all named classes of the application domain, and states whether a concept is intended to have instances (like Country, Province), is a generalization of concepts (like AdministrativeArea as a superclass of related concepts), or an abstraction like Area or Line (similar to an interface in Java) that adds a set of certain properties or constraints to concepts.
- c er:isa er:ReifiedRelationship/er:SymmetricReifiedRelationship and c er:reifies p states whether a concept is a reification of a property.

Annotation Properties will also be used during the ontology inspection for maintaining the correspondences between application classes and ad-hoc classes for testing containment.

The mapping from an ER model to the relational model is rather standard (the basic already being defined in the original paper [1], and by now being taught in every database lecture). Given an OWL ontology (on the class/property level, not containing the instances), one can apply an analogous algorithm to the ontology.

The outcome is the schema itself, plus the mapping information that is used to insert the information of triples into the appropriate tables, tuples, and columns, and for defining a SPARQL-to-SQL mapping for querying.

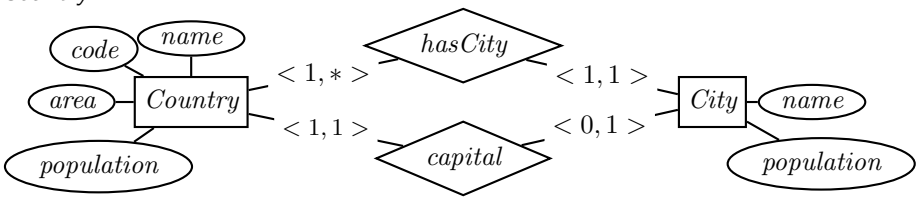
[3] `http://www.dbis.informatik.uni-goettingen.de/Mondial/#RDF`

Structure of the Paper. The paper is structured as follows: Section 2 shortly reviews the main mapping rules for obtaining a relational schema from a conceptual schema. Section 3 describes the algorithm for developing the relational schema from an ontology. Section 4 deals with concretely extracting the appropriate knowledge from the OWL-DL ontology. Section 5 gives a comparison with related work, and Section 6 concludes the paper.

2 Standard Mapping to a Relational Schema

The mapping from an OWL ontology to a relational schema follows the same standard rules as from an ER diagram (cf. e.g. [2, Ch. 7]). The presentation is kept straightforward and informal, omitting some details. This is sufficient to show in Section 4 that rather simple ontology inspection capabilities, e.g., whether a property p is functional on a class c whose intersection with the domain d of p is nonempty, are required that are nevertheless cumbersome tasks given the current languages and tools.

Example 3 *Consider the following fragment from* MONDIAL*: every country has one or more cities, one of which is its capital. Every city is located in exactly one country, but not every city is a capital. The inverse of* hasCity *is named* inCountry*.*



Entity types: Entity types (= non-abstract classes) are mapped to tables where the set of attributes consists of all *functional* attributes. The key attributes are the same as for the ER model (for weak entity types, see textbooks). Since in RDF, for every resource, one of its URIs can be chosen as key (maintaining a same-as table if necessary), there are no weak entity types. For the above example, Country(uri,name,code,population,area) and City(uri,name,population) are such basic table schemas.

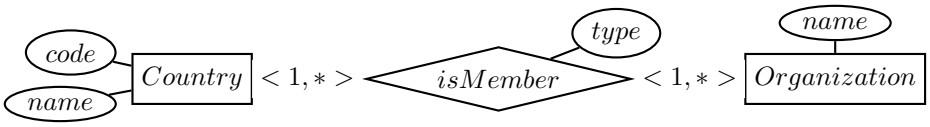
Each multivalued attribute of an entity type is stored in a separate table whose attributes are the keys of the entity relation, and additionally, the attribute name.

Relationship types: The ER model allows arbitrary n-ary relationships. In OWL ontologies, relationships with arity > 2 are always represented by reified classes; thus, the discussion can be confined to binary relations. There, $1:n$- (including 1:1), and $n:m$ relationships have to be distinguished: $1:n$ relationships can be included within the entity table of the n side. E.g., for the capital relationship, the country table is extended to Country(uri,name,code,population,area,capital) while for the hasCity $1:n$-relationship, the inverse inCountry extends the city

table to City(uri,name,population,inCountry,capital-). Note that the 1:1 relationship capital/capital- is at first represented in both directions; one direction can be removed. $n:m$ relationships are mapped to separate tables whose attributes are the keys of the respective entity tables, and the attribute names are chosen according to the entity types (or the roles, in case of recursive relationships) e.g., locatedAt(city,water).

Attributed $n:m$ Relationship types – Reification: Attributes of relationships are added in the same way; but in OWL ontologies, relationships with attributes have also to be modeled by reified classes.

Example 4 *Consider the ER diagram of the isMember relationship between countries and organizations, e.g., Croatia is a candidate member of the EU:*



Here, the OWL ontology models the relationship already as a reified class:

```
:Membership rdfs:subClassOf :ReifiedThing; er:reifies :isMember.
:ofCountry a owl:FunctionalProperty; rdfs:domain :Membership;
  owl:inverseOf :isInMembership; rdfs:range :Country.
:inOrganization a owl:FunctionalProperty; rdfs:domain :Membership;
  owl:inverseOf :hasMembership; rdfs:range :Organization.
:isMember rdfs:domain :Country; rdfs:range :Organization;
  owl:inverseOf :hasMember.
```

Note the use of er:reifies, which is an AnnotationProperty.
The resulting table schema for the ER diagram is isMember(country,organization, type), while from the ontology, the names are slightly different: Membership(ofCountry,inOrganization,type).

Summarizing, all functional properties of entities (including reified ones) are stored in the entity tables, and all multivalued properties are stored in binary tables.

3 The Algorithm

The algorithm conceptually follows the procedure when transforming an ER model to the relational model. The mapping information is stored in a single metadata table that represents all relevant knowledge about the correspondence between classes and their properties, and table names and attributes:

The *Mapping Dictionary* is used to look up where to store an RDF triple (s,p,o), and how to retrieve properties of a resource s (which is required when translating SPARQL queries to SQL). It is a table of the form

$$MD \subseteq (\mathsf{Classes} \times \mathsf{Properties}) \times (\mathsf{Tables} \times \mathsf{Columns} \times \{+,-\})$$

with the semantics that $((c,p),(t,a,+))$ denotes that property p of instances of class c is stored in table t in attribute (=column) a. $((c,p),(t,a,-))$ denotes that the inverse of property p of instances of class c can be found in table t in attribute a.

In the following, p is usually used for functional properties and for multivalued literal properties, and r for $n:m$ properties. p^- and r^- denote the reverse direction.

The mapping algorithm consists of the following steps:

A1: Functional Properties For every non-abstract class c, the primary table schema t_c contains a *uri* column, and columns for all *functional* properties p. Put $((c,p),(t\_c,p,+))$ into the Mapping Dictionary; for object-valued properties with range d, put also $((d,p^-),(t\_c,p,-))$ into the MD.

A2: Correction for 1:1 Properties For each object-valued 1:1-property (i.e., which has been inserted in both directions) p between non-abstract classes c and d ($c=d$ for recursive relationships is possible), one direction is deleted: if there is a $<0,1>$ cardinality (i.e. the property is partial in one direction) on one side and a $<1,1>$ cardinality on the other, delete the direction from the $<0,1>$ side (and delete the MD entries referring to it). Otherwise, arbitrarily delete one of them (see Example 5 below).
(Note that usually there are $1:1$ properties that result from the reification of attributed $1:n$ properties. These are total from the reified side and partial or total from the entity type side – delete the entity side.)

A3: $n:m$ **Relationships** For each $n:m$ property r (do not consider r^- separately, since it is covered by the mapping of r):
- add a new table t_r(dom, rge),
- for each non-abstract class c whose intersection with the domain of r is not empty, put $((c,r),(t\_r,\mathsf{rge},+))$ into the Mapping Dictionary.
- for each non-abstract class d whose intersection with the range of r is not empty, put $((d,r^-),(t\_r,\mathsf{dom},+))$ into the Mapping Dictionary.

Note that instead of the generic column names (dom,rge), any other names can be chosen. A common choice is to use (c,d) where c is the least common superclass that covers the domain (which can be any DL class expression) of r, and d is the least common superclass that covers the range (which can also be any DL class expression) of r. If domain=range, then a common naming is (c_1,c_2) or (c,r) (note that the column names are actually not relevant, because insertions and queries are only made on the RDF level and mapped to the database via the Mapping Dictionary).

A4: Non-Functional Literal-Valued Properties Non-functional literal-valued properties are similar to $n:m$ properties, with the only difference that there is no inverse navigation:
For each non-functional literal-valued property r:
- add a new table t_r(dom, r),
- for each non-abstract class c whose intersection with the domain of r is not empty, put $((c,r),(t\_r,r,+))$ into the Mapping Dictionary.

The above steps are complete for representing the conceptual model.

Theorem 1 *The algorithm is complete and minimal, i.e. every property is finally represented once in the tables, covering both directions in case of object-valued relationships. The Mapping Dictionary is also complete, including the inverses.*

Proof Sketch.

- functional literal-valued properties: Step (A1) handles them. They have no inverse direction.
- 1 : 1 object-valued properties: Step (A1) handles them in both directions (i.e., in both involved class tables); and (A2) deletes one direction.
- non-functional (i.e., multivalued) literal properties: Step (A4) handles them. They have no inverse direction.
- 1 : n and n : 1 object-valued properties: one direction is functional. It is handled in Step (A1).
- $n : m$ properties r: Step (A3) creates an $n : m$ table for r which represents also r^-.

Example 5 (Functional Property and Partial 1:1 Relationship) *Consider again the fragment from* Mondial *given in Example 3. The OWL-DL ontology is as follows: code is a 1:1 literal-valued property; capital is also 1:1, where every country has exactly 1 capital, but not every city is a capital of a country.*

```
:code a owl:FunctionalProperty; a owl:InverseFunctionalProperty;
   rdfs:domain :Country; rdfs:range xsd:string; owl:cardinality 1.
:name a owl:FunctionalProperty; a owl:InverseFunctionalProperty;
   rdfs:domain [owl:UnionOf (:Country :City ...)]; rdfs:range xsd:string;
   owl:cardinality 1.
:capital a owl:FunctionalProperty; a owl:InverseFunctionalProperty;
   rdfs:domain :Country; rdfs:range :City; owl:inverseOf :isCapitalOf;
   owl:cardinality 1.
```

The entries for code in the Mapping Dictionary are as follows:

- *The entry ((Country,code)(t_Country,code,+)) states that for looking up the pattern {:germany a :Country; :code ?C}, the SQL query SELECT code FROM t_Country WHERE uri=':germany' has to be stated.*
- *For looking up the triple pattern {?X a :Country; :code 'D'}, the SQL query SELECT uri FROM t_Country WHERE code='D' has to be stated.*
- *analogously for Country.name and City.name.*

The capital relationship between Countries and Cities is functional in both directions; its inverse is named isCapitalOf. So Step (A1) generates the following:

- *capital is mapped to a column in the t_Country table: t_Country(uri,name,code, capital).*
  The MD entry is ((Country,capital)(t_Country,capital,+)) that states that for looking up the triple pattern {:germany a :Country; :capital ?X}, the SQL

query SELECT capital FROM t_Country WHERE uri=':germany' *has to be stated (yielding the uri binding* ?X/:berlin*).*
The inverse MD entry is ((City,isCapitalOf)(t_Country,capital,-))*, stating that "for looking up* isCapitalOf *of a city (i.e., "of which country is the city with uri* x *the capital?", one can instead lookup for a tuple in the* t_Country *table whose* capital *column has the value* x*. The SPARQL query* {:berlin a :City; :isCapitalOf ?X} *is then mapped to the SQL query* SELECT uri FROM t_Country WHERE capital=':berlin'

– capital- = isCapitalOf *is a functional object-valued property of cities (containing many null values) mapped to a column in the* City *table:* t_City(uri,name, isCapitalOf,...) *yielding the MD entry* ((City,isCapitalOf)(t_City,isCapitalOf,+))*. The inverse MD entry is* ((Country,capital)(t_City,isCapitalOf,-))*.*

As a result so far, the capital/capital- *property would be stored redundantly. Then, Step (A2) removes one direction (namely, the partial one, stored in the* City *table).*

The above example was rather easy, since the properties were globally functional. In general, for each subclass of the domain, functionality has to be checked individually.

4 Ontology Inspection for Schema Generation

To keep the implementation as declarative as possible, it consists of a sequence of SPARQL queries against the TBox ontology, extended with auxiliary definitions. For all non-abstract classes, table schemata as described above are derived where all properties can be stored. If a user intends to tune the mapping, it is easy to merge tables by modifying the Mapping Dictionary accordingly.

4.1 Support for TBox Analysis in SPARQL

SPARQL [3] itself is intended to be a query language for RDF, i.e., for querying triple patterns. Some OWL notions are directly and unambiguously represented by triples, such as c a owl:Class. Some others are directly supported by special handling in the reasoners, e.g., c rdfs:subClassOf d and c owl:equivalentClass d. On the other hand, for instance, from

```
:p a owl:ObjectProperty; rdfs:domain :D.
:D owl:equivalentClass [ a owl:Restriction; owl:onProperty :p; owl:maxCardinality 1 ].
```

p is not contained in the answer to ?P a owl:FunctionalProperty, although this is actually implied by the knowledge base. In the following, even more complex queries are needed, e.g., whether a given property p is functional on some class c that has a non-empty intersection with the domain of p. The following query –if it would be allowed– yields the answers:

```
?C rdfs:subClassOf [ a owl:Restriction; owl:onProperty :p; owl:maxCardinality 1 ].
```

Such functionality is covered by the *SPARQL OWL 2 Direct Semantics Entailment Regime* [4], building upon combining SPARQL with a DL reasoner. Concerning available implementations, a set of atomic built-in notions can be queried by SPARQL-DL [5, 6], but this is not sufficient for the TBox analysis required in the following. Reasoning for the *SPARQL OWL 2 Direct Semantics Entailment Regime* has been presented in [7, 8].

As long as there is no actual implementation available for using SPARQL, instead of a simple declarative query, one has

1. to define dummy classes fcttest_c_p for each relevant class/property combination,
2. add them to the ontology,
3. and answer the query {c rdfs:subClassOf fcttest_c_p}.

4.2 Transformation Steps

In the following, some of the ontology inspection steps are exemplarily described. Again, AnnotationProperties are needed to correlate test classes with application classes and properties. All properties in the aux: namespace are declared to be AnnotationProperties.

4.2.1 Auxiliary: Create Names for Unnamed Inverses

Since in some cases, the inverses are needed as column names in the resulting tables, for all inverses that are not yet named, a synthetic name is created.

Example 6 *Consider that a river may have several estuaries (into some lakes it flows through, and finally into some sea/lake/river). This property hasEstuary is thus inverse-functional, i.e. every estuary belongs to one river. In this step, its inverse hasEstuary_Inv is named and defined, e.g.,*

```
:hasEstuary_Inv a owl:ObjectProperty; a aux:AddedInverseProperty;
  owl:inverseOf :hasEstuary.
```

4.2.2 Identify Functional Properties for each Class

For each non-abstract class c, all properties p that are functional for instances of that class, and for which the intersection of c with the domain of p is non-empty are collected. Note that a functional property may be defined on an abstract class, and may be forbidden by additional restrictions for some concrete subclasses of its domain. Also, properties may be non-functional in general, but specified to be functional for certain subsets of the domain. Thus, simply checking the domain of p, and checking whether p is (globally) functional is not sufficient. Instead, test classes must be defined, added to the ontology, and appropriate queries must be stated:

```
construct {
  [a owl:Restriction; owl:onProperty ?P; owl:maxCardinality 1]
```

```
  aux:isa aux:Card1Test; aux:onDomain ?C; aux:onProperty ?P .
  [a owl:Restriction; owl:onProperty ?P; owl:allValuesFrom owl:Nothing ]
  aux:isa aux:NonEmptyTest; aux:onDomain ?C; aux:onProperty ?P.  }
where
{ ?C er:isa  ?ERC . ?ERC rdfs:subClassOf er:Concrete .
  ?P a rdf:Property; rdfs:domain ?D; rdfs:range ?R.
  FILTER (?P != owl:bottomObjectProperty && ?P != owl:bottomDataProperty)
  # intersection may be non-empty -- (C disjoint D) not provable
  NOT EXISTS { ?C owl:disjointWith ?D }
```

The query constructs the test class definitions for all relevant class/property-pairs.

Example 7 *Geographical things have a name; only estuaries are excluded from this (*Estuary $\sqsubseteq \forall$name.$\bot$*). The property* inCountry *is* $n : m$*, but each administrative area, where cities are a subset of, is located in exactly one country: (*AdmArea $\sqsubseteq \exists 1$inCountry.$\top$*).*

The following are the relevant test classes – note the usage of Annotation-Properties from the aux *namespace:*

```
[] aux:isa aux:NonEmptyTest;
   aux:onDomain :Estuary; aux:onProperty :name;
   a owl:Restriction ; owl:onProperty :name;
                owl:allValuesFrom owl:Nothing  .
[] aux:isa aux:Card1Test;
   aux:onDomain :City; aux:onProperty :inCountry;
   a owl:Restriction; owl:onProperty :inCountry; owl:maxCardinality 1 .
```

The evaluation uses the annotations of the test class definitions, namely by checking whether c is a subclass of the cardinality$\leq$1 test class, and whether c is not a subclass of the (c, p)-Non-Empty-Test class.

```
construct { ?C aux:hasFunctProp ?P }  where
{ ?C1T aux:isa aux:Card1Test; aux:onDomain ?C; aux:onProperty ?P .
  ?NET aux:isa aux:NonEmptyTest; aux:onDomain ?C ; aux:onProperty ?P .
  ?C rdfs:subClassOf ?C1T.
  NOT EXISTS { ?C rdfs:subClassOf ?NET } }
```

Note that better efficiency can be gained when properties that are known to be globally functional (by querying for the explicit triple ?P a owl:FunctionalProperty) bypass the cardinality≤ 1 check.

4.2.3 Further Steps

Further steps are not described in detail here:

– for 1:1 properties remove one direction from the list (if one direction is total, choose this one), and create the Mapping Dictionary entries and the tables for the entity types,

– Note that reified relationships are already covered with this, since they are in $n:1$ relationship with the contributing entity types, cf. Example 4.
– for non-functional attributes and the remaining $n:m$ properties, identify the column names, create the Mapping Dictionary entries and the tables,
– handle symmetric relationships (e.g., mergesWith(sea1,sea2)) and create a view for the symmetric hull.

As a further refinement, it is possible to distinguish between different range subclasses to isolate functional subproperties. For instance, geographical things are usually located in several administrative areas (e.g., some provinces, some countries), but cities are located in exactly one province of exactly one country – thus, for cities, instead of storing this information in an $n:m$ table, two functional properties locatedIn_Country and locatedIn_Province can be added to the city table which makes query evaluation more efficient (note again that the column names are actually not relevant, because insertions and queries are only made on the RDF level and mapped to the database via the Mapping Dictionary). The description of the whole process can be found in [9].

5 Related Work

Several approaches discuss storage strategies for RDF data in relational databases. The efficiency of schema-aware vs. schema-oblivious relational storage representations for RDF wrt. query evaluation has been investigated in [10], with the result that schema-aware representations yield a superior query performance; additionally, different storage representations of subsumption relationships (e.g., implicit vs. explicit) have been considered. This idea has been followed in the *property tables* approach [11] that results in a similar storage layout to the one we have chosen: triples are stored wrt. to a specific subject in the same table (the *functional* property table) which is comparable to our class tables, *multi-valued* properties are stored analogously to our layout for $n:m$ properties. They additionally use a *triple table* for storing the data, which in our approach is not required. The mapping of the RDF data to the relational model is given explicitly and regardless of an available ontology, whereas in our case we automatically derive the storage layout through an analysis of the ontology. The Sesame store [12] supports a dynamic database schema for PostgreSQL[4] where new tables are added based on an analysis of the RDFS schema of an RDF graph. But they do not support a more advanced analysis of the relationships as supported by our ontology analysis. A different approach was taken in [13], where the storage scheme is organized by a *vertical partitioning* on properties, i.e. each table is named after a property and contains exactly two columns, one for the subject and one for the object. In combination with a column-oriented relational DB, typical basic graph pattern queries can be answered by (sort-)merge joins. The downside of this storage organisation is that patterns where the property position contains a variable require a `UNION` over all tables.

[4] `http://www.postgresql.org`

Commercial relational database systems also provide support for storing RDF graphs, e.g., Oracle stores the RDF data in a triples table-style representation where common join combinations can be cached as so-called *subject-property matrix materialized join views*, and queries on RDF graphs can be intermixed with regular SQL tables via a custom `RDF_MATCH` SQL table function [14]. IBM also provides RDF support on top of its DB2 database system[5], where data is stored clustered on common subjects and also in the inverse direction clustered on objects, where the storage layout is optimized based on an analysis of instance data. In our approach the storage layout is determined at design time via ontology inspection, and no further optimization based on instance data is required.

The SP^2Bench benchmark experiment [15] has observed that a *fully relational* DB schema outperforms generic RDF data storage strategies by a factor of typically at least one order of magnitude already for small document sizes. Our approach results in a fully relational DB schema storage schema as known from common ER transformations and as such makes optimal use of the available schema information in an automatic fashion while retaining the appearance of an RDF store to the outside, i.e. regular RDF triples can be inserted. Also, the mapping dictionary gives us the full control of how we store each RDF triple physically with the opportunity to choose different storage layouts, if desired.

Another related aspect is the investigation of SPARQL views over existing relational data. This direction is orthogonal to our approach and can be considered as the inverse direction, i.e. how to use SPARQL queries to access the stored RDF triples. An approach based on a fixed set of mappings is the D2R server [16]. A recent project, called Quest [17], generates optimised SQL queries based on query containment checks, e.g. to eliminate redundant joins. For this, Quest relies on a mapping language that relates the notions of the domain ontology with the relational database. These mappings have to be defined manually by a skilled user. In future work, we want to investigate an integration of the Quest frontend with our storage layout, especially with a focus on the automatic generation of these mapping rules by an investigation of available meta data in our mapping dictionary. For a more complete survey of relational DB to RDF mappings the interested reader is referred to [18].

The $\mathcal{DLR}$ description logic [19, 20] uses general n-ary relations for expressing conceptual models such as the ER model and UML class diagrams. In contrast, in our approach, we *derive* a relational storage layout from a given OWL-DL ontology that has the same properties as the relational model derived from an ER diagram.

6 Conclusion

We have shown a generic and declarative approach to obtain an application-specific relational schema from an OWL-DL ontology. The process is specified

[5] `http://researcher.watson.ibm.com/researcher/view_project.php?id=4416`

completely declaratively by SPARQL queries, using AnnotationProperties for managing correspondences between classes, properties and auxiliary classes for ontology inspection. It cannot be executed inside SPARQL, but requires a sequential external execution to create class definitions by SPARQL queries, and to feed them back into the reasoner before evaluating subsequent queries.

References

1. Chen, P.: The Entity-Relationship Model — Towards a Unified View of Data. ACM TODS **1**(1) (1976) 9–36
2. Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems, 4th Edition. Addison-Wesley-Longman (2004)
3. W3C: SPARQL 1.1 Query Language. `http://www.w3.org/TR/sparql11-query/` (2012)
4. W3C: SPARQL 1.1 Entailment Regimes. `http://www.w3.org/TR/sparql11-entailment/` (2013)
5. Sirin, E., Parsia, B.: SPARQL-DL: SPARQL Query for OWL-DL. OWLED (2007)
6. Clark&Parsia: Pellet: OWL 2 Reasoner for Java. `http://clarkparsia.com/pellet/`
7. Kollia, I., Glimm, B., Horrocks, I.: SPARQL Query Answering over OWL Ontologies. ESWC (2011) 382–396
8. Information Systems Group, University of Oxford: Hermit OWL Reasoner. `http://www.hermit-reasoner.com/`
9. Hornung, T., May, W.: Efficient, Schema-Aware Storage of RDF Graphs. Technical report (2013) `http://www.dbis.informatik.uni-goettingen.de/RDF2Rel/`.
10. Theoharis, Y., Christophides, V., Karvounarakis, G.: Benchmarking Database Representations of RDF/S Stores. ISWC. Springer LNCS 3729 (2005) 685–701
11. Wilkinson, K.: Jena Property Table Implementation. Scalable Semantic Web Knowledge Base Systems. (2006)
12. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. ISWC, Springer LNCS 2342 (2002) 54–68
13. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable Semantic Web Data Management Using Vertical Partitioning. VLDB. (2007) 411–422
14. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An Efficient SQL-based RDF Querying Scheme. VLDB. (2005) 1216–1227
15. Schmidt, M., Hornung, T., Küchlin, N., Lausen, G., Pinkel, C.: An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario. ISWC. Springer LNCS 5318 (2008) 82–97
16. Bizer, C.: D2R MAP — A Database to RDF Mapping Language. WWW (2003)
17. Rodriguez-Muro, M., Hardi, J., Calvanese, D.: Quest: Efficient SPARQL-to-SQL for RDF and OWL. ISWC Posters & Demos. Vol. 914 of CEUR Workshop Proceedings. CEUR-WS.org (2012)
18. Sahoo, S.S., Halb, W., Idehen, S.H.K., Jr, T.T., Auer, S., Sequeda, J., Ezzat, A.: A Survey of Current Approaches for Mapping of Relational Databases to RDF. Technical report, W3C RDB2RDF Incubator Group (January 2009)
19. Calvanese, D., Lenzerini, M., Nardi, D.: Description Logics for Conceptual Data Modeling. Logics for Databases and Information Systems, Kluwer (1998) 229–263
20. Cali, A., Calvanese, D., Giacomo, G.D., Lenzerini, M.: A Formal Framework for Reasoning in UML Class Diagrams. ISMIS. Springer LNAI 2366 (2002) 503–513