Efficient, Schema-Aware Storage of RDF Graphs

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

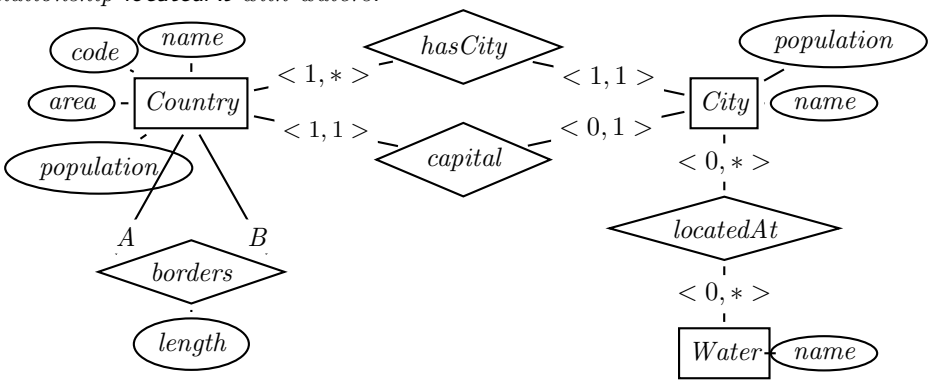
OWL Ontologies (sometimes) provide comprehensive knowledge about the concept level of an application: characterizations of the classes of relevant resources, the class hierarchy, properties, and relationships between them. Such information can be used, e.g., for designing a relational storage schema.

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:

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

Example 1 (ER Model) *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*. Countries are in a (symmetric)* borders *relationship with each other; the relationship has a* length *attribute. Cities are in an* $n : m$ *relationship* locatedAt *with waters.*



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.*

Amongst Waters*, there are* Seas *and* Lakes*, both are* Areas*, and* Rivers*, which are* Lines*.*

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*).*

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 the same 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.

2 Standard Mapping to a Relational Schema

The mapping from an OWL ontology to the 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 8 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.

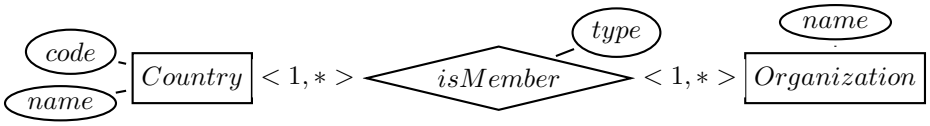
Entity types: Entity types (= non-abstract classes) are mapped to tables where the set of attributes consists of all *functional* attributes. The key attributes are the same as for the ER model (for weak entity types, see textbooks). Since in RDF, for every resource, one of its URIs can be chosen as key (maintaining a same-as table if necessary), there are no weak entity types. For instance, 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 into 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 3 *Consider the ER diagram of the isMember relationship between countries and organization, 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.
:ofMember 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(ofMember, inOrganization, type)*.*

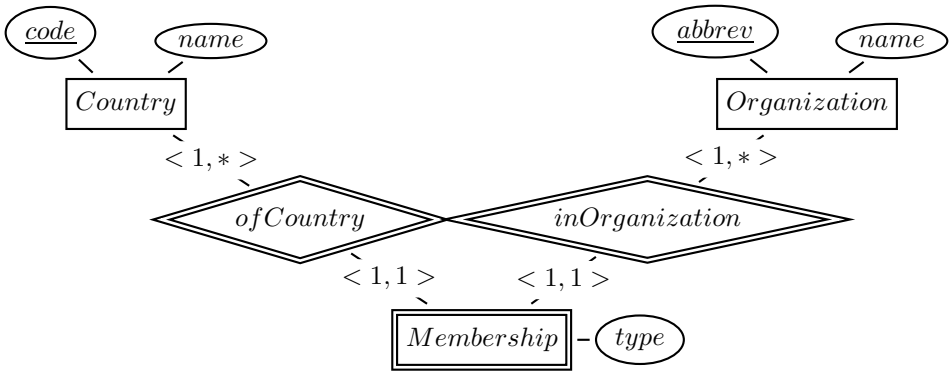
Summarizing, all functional properties of entities (including reified ones) are stored in the entity tables, and all multivalued properties are stored in binary tables.

2.1 Conceptual Modeling Notions

The oldest, and most prominent conceptual model is the ER model [1]. RDFS is strictly weaker in its modeling notions. There is e.g. no notion of attributed relationships or n-ary relationships (in the same way as for the CODASYL Network Data Model and for UML). Missing notions have to be expressed by reification. Note that in RDFS, such relationships are then expressed by the reification entity type and (at least) $n + m$ properties (where m is the number of attributes of the relationship).

Example 4 *Consider again the ER diagram of the* isMember *relationship from Example 3.*

The translation into binary, non-attributed relationships requires a reification of isMember *into an entity type* Membership *with a single attribute* type *(*Membership *is a weak entity type, identified by its relationships to a country and an organization). The reification entity type is in a* $< 1, 1 >$ *relationship complexity wrt. each of the new relations.*



The resulting table schema for the original ER diagram is isMember(country, organization, type)*, while for the reified modeling, the names are slightly different:* Membership(ofMember, inOrganization, type)*.*

The encoding of n-ary, $n > 2$, relationships is similar. As an aside, note that the expressiveness wrt. the description of relationship complexities is completely retained in the reified modeling (via the "outer" complexities).

2.2 Possible Metadata Description Formats

Dependent from the intended usage, the "starting point" metadata is either an ER model, or an RDFS/OWL description of an already existing RDF database.

2.2.1 Usage of Annotation Properties In OWL-DL, classes and properties are not allowed to have "normal" properties. OWL-DL provides a set of distinguished properties (some of them taken from RDFS) that are allowed to be applied to classes and properties.

For making further statements about classes and properties, OWL-DL supports the notion of *Annotation Properties*. Note that Annotation Properties themselves are not allowed to have further built-in properties like rdfs:domain, rdfs:range, or owl:inverseOf.

In our approach, Annotation Properties will be used for further describing the ontology itself, and also during the ontology inspection for maintaining the relationship between application classes and ad-hoc classes for testing containment.

2.2.2 RDFS or (RDFS and OWL): If the metadata description is given in RDF, there are two possibilities:

1. pure RDFS, i.e. without cardinalities,
2. OWL, with cardinalities and possibly keys.

In case of (2), via reification, most of the expressiveness of the ER model is formally available; nevertheless, the information about reification itself is missing. In case (1), without cardinalities, important information for an optimal (and optimally efficient) mapping to the relational model is missing.

2.2.3 Describing ER Models in RDF Since the ER model is a graphical model, it can only be used when encoded into some machine-readable form.

Below, we give an RDF vocabulary that (straightforwardly) encodes the main modeling notions of ER models. The meaning of Class, subClassOf, Property, domain and range is (nearly) as in RDFS, and cardinality, {min|max }Cardinality is as in OWL, with the extension that "*" is allowed as maximal cardinality. Additionally, there is AttributedProperty and ReifiedProperty (for reification as in ER aggregation). hasAttributes is used for enumerating allowed attributes; hasKey and inverseOf have the same semantics as in OWL. The language uses the er: namespace. All properties in the er: namespace are declared as owl:AnnotationProperties

Example 5 *Consider again the situation given in Example 3.*
In RDFS together with OWL, this is expressed as follows:

```
:Membership a owl:Class;
  owl:hasKey (:ofCountry :inOrganization).
:ofCountry rdfs:domain :Membership; rdfs:range :Country;
  owl:cardinality 1; owl:inverseOf :isInMembership.
:inOrganization rdfs:domain :Membership; rdfs:range Organization;
  owl:cardinality 1; owl:inverseOf :hasMembership;
  owl:inverseOf [ owl:minCardinality 2 ]. -- every org has at least two members
:type rdfs:domain :Membership; rdfs:range xsd:string;
  rdfs:subClassOf [ a owl:Restriction; owl:onProperty :type;
                    owl:cardinality 1 ] .
:hasMember rdfs:domain :Organization; rdfs:range :Country;
  owl:inverseOf :hasMember.
```

It cannot be expressed in OWL that Membership is not a "usual" entity type, but a reified relationship (reification of :hasMember). Although, classes that participate only $<1,1>$ *in each of its relationships and do not have a proper key are usually reified relationships.*

Thus, the use of Annotation Properties, allows to specify additional knowledge about the ontology:

```
:hasMember er:isa er:AttributedProperty; er:reifiedBy :Membership.
:Membership rdfs:subClassOf :ReifiedThing; er:reifies :hasMember.
```

This knowledge can e.g. be used to define :hasMember as a view over :ofCountry and inOrganization, equivalent to the property chain specification

$$:\text{hasMember} \equiv (:\text{inOrganization-} \circ :\text{ofCountry}) .$$

2.2.4 Further Annotations on Classes and 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.

2.3 Ontology Inspection in SPARQL

The required features for ontology inspection are not supported by SPARQL queries against the ontology in a direct way. This will be discussed in Section 8.1.

2.4 Knowledge about Class-Membership

When using any semantical or conceptual data model (instead of a simple syntactical triple store), the class membership of objects is important.

The following terminology is used:

Class-Definite: At least after inserting all data, for each described resource/object/entity, it is known of which unique non-abstract class it is a member. For simplicity, we assume that in this case, for each resource/object/entity, an explicit tuple (s rdf:type c) has been processed (w.l.o.g.; otherwise an OWL Reasoner must process objects that are not explicitly assigned to a class)

Non-Class-Definite: After inserting all data, there may be objects for which class-membership is not known. Obviously, they cannot be assigned entity types; thus information about them remains in a vertical partitioning storage model.

3 Mapping to the Relational Model

In cases that an ER model, or an RDFS+OWL description (including cardinalities) is given, the ER model can already be finalized from the metadata description. In case that only an RDFS model (without cardinalities) is given, the actual cardinalities must be discovered (see Section 9).

Recall from the standard ER-to-Relational transformation [2, Ch. 7] that all functional properties are stored within the entity tables. Multivalued attributes and $n : m$ relationships are stored in separate tables, while $1 : n$ and $m : 1$ relationships are stored within the entity table for the range and domain class, respectively.

This section describes a basic mapping from the modeling perspective. Section 4 introduces the metadata tables for the operational handling. Section 5 then shows more detailed mappings for special cases, and for better performance for query evaluation.

3.1 Deriving the Relational Model

The result of this algorithm is a relational schema, and a *Mapping Dictionary* which encodes the mapping Ψ

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

with the semantics that $((c, p), (t, a, +))$ denotes that property p of instances of class c is stored in table t in attribute 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.

Comment *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.*

A1: Functional Properties For every class c, the primary table schema t_c contains a *uri* column, and columns for all *functional* properties p (here, functional means that the property is *globally* functional, i.e., for every class in its domain). 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.

(Note that all object-valued properties have to be considered in both directions as p and p^- (inverse)).

A2: Correction for 1:1 Properties For each object-valued 1:1-property (i.e., which has been inserted in both directions) p between 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 (cf. Example 7 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 $t\_r(\mathsf{dom}, \mathsf{rge})$,
- 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). The table is used only internally, and the used naming is stored in 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 $t\_r(\mathsf{dom}, \mathsf{r})$,
- 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.

A5: The above steps are complete for representing the conceptual model in the class-definite case.

For the non-class-definite case, or for incrementally inserting tuples, TOP tables as described later in Section 3.5 are also needed for every functional property whose domain includes more than one class.

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 (and the Mapping Dictionary is also complete, including the inverses):*

- *functional literal-valued properties: Step (A1) handles them. 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^-*.*

Comment *The current processing (cf. Section 8) of mondial-meta requires the inverse to be named explicitly (*flowsThrough-*) to detect functionality.*

3.2 Reference Attributes/Join Attributes

The foreign keys act as reference attributes, and thus provide the join attributes for SQL queries, are implicit, but can be derived from the given data:

a) join object-valued functional properties $c.p$ (e.g., Country.capital) with domain d with range class tables: There is an MD entry of the form $((c,p)(t\_c,p',+))$ (primes denote possible renaming). Then, $t\_c.p'$ refers to the values of $t\_d.id$.
b) Join $n:m$ tables with domain class tables: $n:m$ tables have two attributes. For a Mapping Dictionary Entry $((c,p),(t\_p,d',+))$ where t_p has attributes d' and c', $t\_p.d'$ refers to the class table attribute $t\_d.id$ of the target class d of $c.p$. One can conclude that $t\_p.c'$ refers to $t\_c.id$ and serves as join attribute.
– the above generalizes to the fact that several MD entries match.

In Sections 4.3 (range classes) and 4.4 ($n:m$ relationships) an instantiation of corresponding metadata tables will be given.

3.3 Semantics and Inverse Direction in the Mapping Dictionary

The entries for the inverse directions of object-valued properties are defined as follows:

- Given an entry for a functional object-valued property p of class c with range d, the MD entry is $(c,p),(t\_c,p',+)$ (primes denote possible renamings). The entry for its inverse $d.p^-$ is $(d,p^-),(t\_c,p',-)$.
  - "Look up $x.p$ for $x \in c$ by selecting column p' in table t_c of the tuple describing x and then use the obtained id value for looking up its properties in t_d (by $t\_c.p' = t\_d.id$) or in some t_r (by $t\_r.d'' = t\_c.p'$)."

- "Look up $y.p^-$ for $y \in d$ by selecting the ids of the tuple(s) in t_c s.t. $t\_c.p' = y$."

– Given an entry for an $n : m$ property r between classes c and d (assume the direction r to be stored in $t\_r(c', d')$). The primary MD entry is $((c, r), (t\_r, d', +))$:
  - "Look up $x.r$ by joining $t\_c.id = t\_r.c'$ and use the value $t\_r.d'$, e.g., for joining $t\_r.d' = t\_d.id$."

  The entry for its inverse is $((d, r^-), (t\_r, c', +))$:
  - "Look up $y.r^-$ by joining $t\_d.id = t\_r.d'$ and use the value of $t\_r.c'$ (for getting the ids of the corresponding x values)".

  Note that the latter entry is equivalent to $((d, r^-), (t\_r, d', -))$:
  - "Look up $y.r^-$ by joining $t\_d.id = t\_r.d$ and use the value of $t\_r.c$ (for getting the ids of the corresponding x values)".

  $\Rightarrow$ for $n : m$ tables, no entries with "Inv='-'" are needed. "Inv='-'" is only needed for navigation "into" class tables (including reified property ones). This is discussed in Section 4.7.
– Note that however $c \to c'$ and $d \to d'$ are renamed, the MD entry is sufficient to know one of the column names in t_r, and to determine the other from the table header/data dictionary.

3.4 Examples

Example 6 *Consider an RDF graph with a class c with a literal-valued functional property p, a functional object-valued property q, and an object-valued $n : m$ property r with range d.*

Then, the table t_c has columns id, p, q. The table t_r has columns c and d. The Mapping Dictionary contains the entries $((c, p), (t\_c, p, +))$, $((c, q), (t\_c, q, +))$, $((d, q^-), (t\_c, q, -))$, $((c, r), (t\_r, d, +))$, and $((d, r^-), (t\_r, c, +))$.
The SPARQL query pattern

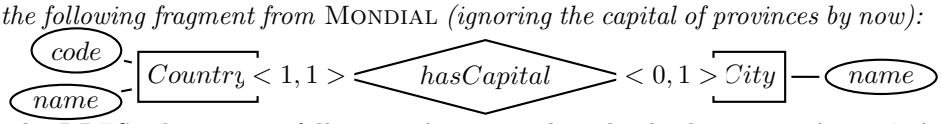
```
select ?O ?V ?W
where { ?X a c; p ?O ; q ?V ; r ?W }
```

(returning literals for O and V, and a URI for W) is mapped to the SQL query

```
SELECT t_c.p, t_c.q, t_r.d
FROM t_c, t_r
WHERE t_c.id = t_r.c
```

The separated relationship tables are always used ("navigated") in a join from an entity table of some class (here: c). The join condition is thus always $t\_c.id = t\_r.c$ according to the column naming in t_r.

Example 7 (Functional Property and Partial 1:1 Relationship) *Consider the following fragment from* MONDIAL *(ignoring the capital of provinces by now):*



The RDFS schema is as follows: code is a 1:1 literal-valued property; hascapital 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.
:hasCapital a owl:FunctionalProperty; owl:inverseOf :isCapitalOf;
rdfs:domain :Country; rdfs:range :City.
:isCapitalOf a owl:FunctionalProperty; owl:minCardinality 0.

The entries of the Mapping Dictionary Ψ are as follows:
First, the entries for code are discussed. code is a string-valued column in the t_Country table.

- *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 id=':germany' has to be stated.*
- *For looking up the triple pattern {?X a :Country; :code 'D'}, the SQL query SELECT id 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. So Step (A1) generates the following:

- *hasCapital is a functional reference property of countries, mapped to a column in the t_Country table: t_Country(id,name,code,hasCapital).*
  The MD entry is ((Country,hasCapital)(t_Country,hasCapital,+)) that states that for looking up the triple pattern {:germany a :Country; :hasCapital ?X}, the SQL query SELECT hasCapital FROM t_Country WHERE id=':germany' has to be stated (yielding the uri binding ?X/:berlin).
  The inverse MD entry is ((City,hasCapital$^-$)(t_Country,hasCapital,-)). stating that "for looking up hasCapital-$^-$ of a city (i.e., "of which country is the city with id x the capital?", one can instead lookup for a tuple in the t_Country table whose hasCapital column has the value x. Since the inverse is named, the entry is equivalent to ((City,isCapitalOf)(t_Country,hasCapital,-)). The SPARQL query {:berlin a :City; :isCapitalOf ?X} is then mapped to the SQL query SELECT id FROM t_Country WHERE hasCapitals=':berlin'
- *hasCapital- = isCapitalOf is a functional reference property of cities (where null values occur frequently) mapped to a column in the City table: t_City(id,name,isCapitalOf,...) yielding the MD entry ((City,isCapitalOf)(t_City,isCapitalOf,-)). The inverse MD entry is ((Country,hasCapital)(t_City,isCapitalOf,-)).*

As a result so far, the property would be stored redundantly. Then, Step (A2) removes one direction. Since Country.hasCapital is (nearly) total and City.isCapitalOf is really very partial, i.e., contains lots of null values, t_City.isCapitalOf and the two MD entries referring to it are removed.

Example 8 (Multi-Domain Functional Relationships) *The hasCapital relationship is also a functional property of provinces. This means, City.isCapitalOf*

is actually not functional, but functional wrt. countries and functional wrt. provinces (in reality, a city may even be capital of several provinces, but since this direction has been removed above, this is no problem. (Details how to refine the mapping of a relationship that is not functional in general, but wrt. specific non-abstract classes as ranges are discussed in Sections 5.1 and 5.6.)

So, there are also the MD entries ((Province,hasCapital)(t_Province,hasCapital,+)) and ((City,isCapitalOf)(t_Province,hasCapital,-)). Note that with this, for (City,isCapitalOf), there are two entries – for being capitals of countries and provinces. If the range is undetermined, a query must obviously use the union of transforming both of them:
The SPARQL query

```
select ?N
where { ?C a :City; :name "Rome"; :isCapitalOf ?X . ?X :name ?N. }
```

where the searches class of ?X is not known must look up both tables to return e.g. ?N/"Italy" and ?N/"Lazio":

```
(SELECT x.name
FROM t_City c, t_Country x
WHERE c.name="Rome" AND c.id = x.capital)
UNION
(SELECT x.name
FROM t_City c, t_Province x
WHERE c.name="Rome" AND c.id = x.capital)
```

Even when the range class is determined

```
select ?N
where { ?C a :City; :name "Rome"; :isCapitalOf ?X .
        ?X a :Province; :name ?N . }   ,
```

the information in the plain Mapping Dictionary is not sufficient for restricting the lookup to the province table. For this reason, the Mapping Table used in the process will provide additional information (cf. Section 4).

In Section 5.6, another example will be shown that demonstrates a refinement where the range specification is not only useful when considering the inverse direction.

Example 9 (Reified Relationships) *Recall the attributed relationship and its reification from Example 3.*

The primary table schemata after step (A1) are Country(id,code,name,population,area), Organization(id,abbrev,name,established), and Membership(id,type) (the Membership instances may be blank nodes that have only local uris).

The properties isInMembership and hasMembership are $1:n$ (so their inverses ofCountry and inOrganization are functional). Each country and each organization may be related to multiple Membership instances. Step (A1) adds the relationships in their functional direction, both to the t_Membership table:

Membership(id,ofCountry,inOrganization,type)
The table Membership(id,ofCountry,inOrganization,type) has the same form as the isMember(country,organization,type) table that is obtained when transforming the original ER diagram directly into the relational model.
The Mapping Dictionary then contains
((Country,code),(t_Country,code,+)),
((Country,name),(t_Country,name,+)),
((Country,isInMembership),(t_Membership,ofCountry,-)),
((Organization,abbrev),(t_Organization,abbrev,+)),
((Organization,name),(t_Organization,name,+)),
((Membership,ofCountry),(t_Membership,ofCountry,+)),
((Membership,inOrganization),(t_Membership,inOrganization,+)),
((Membership,type),(t_Membership,type,+)).
The SPARQL query

```
select ?N ?A ?T
where { ?X a :Country; :name ?N; :isInMembership ?M .
        ?M :inOrganization ?O ; :type ?T . ?O :abbrev ?A }
```

is mapped to the SQL query using ("lookup ?X :isInMemberhip ?M by finding its inverse in the ofCountry column of t_Membership")

```
SELECT t_Country.name, t_Membership.type, t_Organization.abbrev
FROM t_Country, t_Membership, t_Organization
WHERE t_Country.id = t_Membership.ofCountry
AND t_Membership.inOrganization = t_Organization.id
```

Note that the columns corresponding to object-valued properties are only renamed when the property is separated into an own table. For this reason, in the above example there is no renaming in the final tables.

Example 10 (n:m relationship between non-abstract classes) *Consider the $n:m$ membership relationship between countries and organizations (ignoring the membership type). The additional RDFS tuples are*

```
:Organization a owl:Class.
## organizations have a functional attribute name, and ...
:ismember rdfs:domain :Country; rdfs:range :Organization;
  owl:minCardinality 0; owl:maxCardinality *;
  owl:inverseOf [ owl:minCardinality 0; owl:maxCardinality * ] .
```

Primary tables created by Step (A1) are the class tables t_Country(id,name,code,...) and t_Organization(id,name,abbrev,headq,...).
Step (A3) maps the $n:m$ property ismember to a new table t_ismember(Country,Organization) (applying the column renaming) with the MD entries
((Country,isMember),(t_isMember,Organization,+)),
((Organization,hasMember),(t_isMember,Country,+)).
The SPARQL query

```
select ?CN ?ON
where { ?X a :Country; name ?CN; :ismember ?O . ?O :name ?ON }
```

is mapped to the SQL query

```
SELECT t_Country.name, t_Organization.name
FROM t_Country, t_isMember, t_Organization
WHERE t_Country.id = t_isMember.Country
AND t_isMember.Organization = t_Organization.id .
```

Analogously,

```
select ?ON ?CN
where { ?O a :Organization; name ?ON; :hasmember ?C . ?C :name ?CN }
```

is mapped to the SQL query

```
SELECT t_Organization.name, t_Country.name
FROM t_Organization, t_isMember, t_Country
WHERE t_Organization.id = t_isMember.Organization
AND t_isMember.Country = t_Country.id
```

Note that in both above SQL queries, there are join conditions between t_Country, t_isMember, and t_Organization.

Example 11 (n:m relationship between abstract classes) *Consider the locatedIn $n:m$ relationship between all kinds of "small" geographical things, e.g., cities, mountains, rivers, lakes and all kinds of areas, e.g., provinces, countries, deserts, continents. The domain of locatedIn is an abstract, even unnamed, class (in Mondial defined as the intersection of GeographicalThing with the complement of Continent), and the range is the abstract class Area.*

Step (A3) creates the table t_locatedIn(GeoThing,Area) and the Mapping Dictionary entries

((:City,:locatedIn),(t_locatedIn,Area,+)),
((:Country,:locatedIn),(t_locatedIn,Area,+)),
((:Mountain,:locatedIn),(t_locatedIn,Area,+)),
((:River,:locatedIn),(t_locatedIn,Area,+)),
((:Country,:locatedIn$^-$),(t_locatedIn,GeoThing,+)),
((:Continent,:locatedIn$^-$),(t_locatedIn,GeoThing,+)),
((:Sea,:locatedIn$^-$),(t_locatedIn,GeoThing,+)),
etc.

3.5 TOP Tables

If the database should be used for incrementally storing RDF triples, even in the class-definite case, it cannot be assumed that when a triple (s p o) about a property of some subject is inserted, the class of the subject is already known; e.g. (o123 :name "Llullaillaco"), where instances of several classes have a name

property. For this, binary TOP_p tables are used; e.g., TOP_name(oid,name) which store one property each (it is not known what properties occur together for such an object). The same holds e.g. for population and elevation, and also for object-valued functional properties like capital. (e.g.; (o123 :elevation 6734) – since cities and mountains and some other things have a name and an elevation; one does still not know what "Llullailaco" is; it even might have a population and/or a capital).

In the class-definite case, the TOP tables are empty after the whole input is processed.

Nevertheless, since the TOP tables are usually required during processing the input, they have to be incorporated into the Mapping Dictionary as follows.

The TOP tables for the above (functional) properties have the following Mapping Dictionary entries (this portion of the database is equivalent to vertical partitioning):

((<unknown>,:name),(TOP_name,name,+)) ,
((<unknown>,:population),(TOP_population,population,+)) ,
((<unknown>,:elevation),(TOP_elevation,elevation,+)) .
((<unknown>,:capital),(TOP_capital,capital,+)) .

Note that since capital is object-valued, it has an inverse (whose domain is the City class) for which the TOP table must also be listed in the MD:

((:City,:capital-),(TOP_capital,capital,-)) .

TOP tables are *necessary* for all functional properties whose domain overlaps/contains more than one class. For functional properties whose domain is only a single class, the use is optional (cf. discussion in Section 5.8).

For obtaining data from several TOP tables, a join via oid is used:

```
SELECT n.name, e.elevation
FROM TOP_name n, TOP_elevation e
WHERE n.oid = e.oid;
```

Get the name of the thing (whose class is still unknown) where Paris is the capital of:

```
SELECT c.name, tn.name
FROM t_City c, TOP_capital tc, TOP_name tn
WHERE c.name= "Paris"
    AND c.oid = tc.capital
    AND tc.oid = tn.oid .
```

Since $n : m$ tables are independent from the domain class, there is no separate TOP table for them. Nevertheless, MD entries that deal with the insertion of triples with unknown subject class or object class are required:

((<unknown>,:locatedIn),(locatedIn,Area,+)) ,
((<unknown>,:locatedIn-),(locatedIn,GeoThing,+))

3.6 Object-Class-Dictionary

The *object-class (OC) dictionary* holds for each object id/uri the class where the object is a member of. Its schema is OC(object,class).

4 Metadata Tables

The previous section just discussed the abstract modeling and the Mapping Dictionary which contains all necessary information about the mapping.

This section introduces additional, *derived* tables that (i) make correspondences and meta information more explicit and (ii) are useful for transformation of SPARQL queries to SQL queries.

For this section, consider the following schema:

- Cities: have a name, a population, yielding the following main table:
  t_City: id, name, population, ...
  Additionally, they can be capitals of countries and provinces.
- Countries: have a name, a population, and a capital; yielding the following main table:
  t_Country: id, name, population, capital, ...
- Provinces: have a name, a population, and a capital; yielding the following main table:
  t_Province: id, name, population, capital, ...
- Population and capital has a TOP table:
  TOP_population: id, population TOP_capital: id, capital
- Continents have a name and an area, yielding the main table
  t_Continent: name, area
- A mountain has a name and an elevation yielding the following main table:
  t_Mountain: id, name, elevation, ...,
- Cities and other geographical things can be located on one or more islands, which yield an $n : m$ table:
  t_locatedOnIsland: GeoThing, Island
  (Note: locatedOnIsland is functional for mountains, but $n : m$ for cities! This will be discussed in Section 5.4).
- for *locatedIn*, there is also only a single $n : m$ table (renaming the columns with the least common superclasses):
  t_locatedIn: GeoThing, Area
- EncompBy(countryEncBy$^-$,encCont,percent) is a reified relationship between countries (Country-countryEncBy→EncompBy) and continents (EncompBy-encCont→Continent) with a percent attribute.

4.1 Mapping Dictionary

The Mapping Dictionary

$$MD \subset (\text{Classes} \times \text{Properties}) \times (\text{Tables} \times \text{Columns} \times \{+, -\})$$

is represented as a table (*Class, Property, Range, Table, Column, Inv)* with an additional range column (entries in parentheses for the Range column are optional) with the following sample entries:

MappingDict					
Class	**Property**	**Range**	**Table**	**LookupProp**	**Inv(+-)**
:City	:name		t_City	name	+
:Country	:name		t_Country	name	+
:Continent	:name		t_Country	name	+
<unknown>	:name		TOP_name	name	+
:City	:population		t_City	population	+
:Country	:population		t_Country	population	+
<unknown>	:population		TOP_population	population	+
:Country	:area		t_Country	area	+
:Continent	:area		t_Continent	area	+
<unknown>	:area		TOP_area	area	+
:Country	:capital	(:City)	t_Country	capital	+
:Province	:capital	(:City)	t_Province	capital	+
<unknown>	:capital	(:City)	TOP_capital	capital	+
:City	:capital-	:Country	t_Country	capital	-
:City	:capital-	:Province	t_Province	capital	-
:City	:capital-	<unknown>	TOP_capital	capital	-
:City	:locOnIsland	(:Island)	t_locOnIsland	Island	+
:Mountain	:locOnIsland	(:Island)	t_locOnIsland	Island	+
<unknown>	:locOnIsland	(:Island)	t_locOnIsland	Island	+
:Island	:locOnIsland-	(:GeoThing)	t_locOnIsland	GeoThing	+
:City	:locatedIn	(:Area)	t_locatedIn	Area	+
:Mountain	:locatedIn	(:Area)	t_locatedIn	Area	+
:River	:locatedIn	(:Area)	t_locatedIn	Area	+
<unknown>	:locatedIn	(:Area)	t_locatedIn	Area	+
:Province	:locatedIn-	(:Area)	t_locatedIn	GeoThing	+
:Country	:locatedIn-	(:Area)	t_locatedIn	GeoThing	+
:Continent	:locatedIn-	(:Area)	t_locatedIn	GeoThing	+
(:Area)	:locatedIn-	(:Area)	t_locatedIn	GeoThing	+
:EncompBy	:countryEncBy-	(:Country)	t_EncompBy	countryEncBy$^-$	+
:EncompBy	:encCont	(:Continent)	t_EncompBy	encCont	+
:Country	:countryEncBy	(:EncompBy)	t_EncompBy	countryEncBy$^-$	-
:Continent	:encCont-	(:EncompBy)	t_EncompBy	encCont	-

Its semantics is

> "Property *Prop* of instances of class *Class* with range *Range* is stored in property *lookupProp* in table *table*. Use it as it is ("+"), or inversely ("-")". Entries in parentheses denote that there are no alternative ranges – means "for all applicable ranges".

Multiple Entries. In case that for a given pair (*Class, Prop*), multiple entries are given, then in general, the semantics is disjunctive: *all* entries have to be looked up. If the range column contains a value and the query is specified insofar that the range class is known, then, *only* the matching entry has to be looked up (cf. Section 5.6).

The MD is also used for deciding where to *store* a property. Thus, (*Class*, *Property*, *Range*) must be unique to prevent ambiguities. With the standard transformation discussed in Sections 3.1–5, this is guaranteed.

When considering additional subclassing issues in Section 6, this will not longer be the case. Then, for multiple entries, W/R (Write – unambiguous, and Read - disjunctive/union) must be annotated.

Example 12 *SPARQL queries are translated as follows using the mapping dictionary information:*

```
?C a :Country; :name ?N :population ?P.

SELECT name, population
FROM t_Country;

?C a :Country; :name ?CN; :capital ?Cap.
?Cap a :City; :name ?CapN.

SELECT c.name, cty.name
FROM t_Country c, t_City cty
WHERE c.capital =    -- the join condition cannot be derived from the MT
    -- since the class of the target is known, it must be the id of the class
table:
c.capital = cty.id;

?Cap a :City; :name ?CapN; :isCapitalof ?C.
?C a :Country; :name ?CN.

SELECT cty.name, cname
FROM t_City cty, t_Country c
WHERE c.capital = cty.id;    -- similar to above

?C a :City; :name ?CN :locatedOnIsland ? I.
?I a :Island; :name ?IN

SELECT c.name, i.name
FROM t_City c, t_locOnIsland l, t_Island i
WHERE    -- the join conditions cannot be derived from the MT
    -- but it must be "the other" column of the n : m table:
    l.GeoThing = City.id    -- from entry (10)
    AND l.Island = Island.id;    -- from entry (12)
```

The latter step is made explicit by the metadata table which is introduced in Section 4.3.

4.2 Auxiliary Tables

In the following, views are defined for metadata tables. These depend on the type of tables, i.e., if it is a class table or an $n : m$ table. Thusm ausxiliary tables are used just to keep lists of table names:

NMTables
TableName
mergesWith
locatedIn
locatedAt
locatedOnIsland
locatedInWater

4.3 Range Class Join Attributes

A first step towards the join conditions is the following table which contains all foreign-key-to-primary-key (which is always the id column) referential integrity constraints:
The Range Class Join Attributes (RC) table is a mapping

$$\text{Table} \times \text{Column} \to \text{RangeTable} .$$

The semantics is "the values of column *column* of table *table* refer to the ids of the class table which is given in *RangeTable*":

RangeClassJoinAttrs		
Table	**Column**	**RangeTable**
t_Country	capital	t_City
t_Province	capital	t_City
TOP_capital	id	t_Province
TOP_capital	id	t_Country
TOP_capital	capital	t_City
t_locOnIsland	GeoThing	t_City
t_locOnIsland	GeoThing	t_Mountain
t_locOnIsland	Island	t_Island
t_locatedIn	GeoThing	t_City
t_locatedIn	GeoThing	t_Mountain
t_locatedIn	GeoThing	t_River
t_locatedIn	Area	t_Province
t_locatedIn	Area	t_Country
t_locatedIn	Area	t_Continent
t_EncompBy	countryEncBy-	t_Country
t_EncompBy	encCont	t_Continent

The table can be created and filled by

```
CREATE TABLE RangeClassJoinAttrs
AS
(SELECT md1.Table, md1.LookupProp, classtablename(md2.Class)
 FROM MappingDict md1, mappingDict md2
 WHERE md1.Property is inverse of md2.Property)
 -- md1(Country,capital) x md2(:City,capital-)   -> Country,capital,City
```

```
-- md1(locatedIn,Area)  x md2(:Prov,locatedIn-) -> Province,locatedIn,Province
-- md1(locatedIn,Area)  x md2(:Country,locatedIn-) -> Country,locatedIn,Country
-- md1(locatedIn,Area)  x md2(:Continent,locatedIn-) -> Continent,locatedIn,Country
-- md1(locatedIn,Area)  x md2(:Continent,locatedIn-) ->  --
-- TODO: check for WATER -> River/Lake/Sea etc
```

Note that the MD entries for <unknown> (i.e., for the TOP or most general tables) do not yield entries in the RC table. If instead of <unknown>, an abstract class is given in the MD (cf. discussion in Section 6) that has a class table, the above query generates appropriate entries for the RC table.
The join to be added to the query is *Table.Column* = *RangeTab*.id.

Example 13 *The "capital" and "locatedOnIsland" queries from Example 12 can then be written:*

```
?C a :Country; :name ?CN; :capital ?Cap.
?Cap a :City; :name ?CapN.

SELECT c.name, cty.name
FROM t_Country c, t_City cty
WHERE c.capital = cty.id;    -- from entry (1)

?C a :City; :name ?CN :locatedOnIsland ? I.
?I a :Island; :name ?IN

SELECT c.name, i.name
FROM t_City c, locOnIsland l, t_Island i
WHERE    -- the second join condition can now be written:
    AND l.Island = Island.id    -- from entry (8)
    -- but the first join condition is still implicit and must be
    --"the other" column of the n : m table:
    AND l.City = City.id;
```

For $n : m$ tables between non-abstract classes, usually, the *column* name coincides with the *RangeTab* name. This is not the case for recursive relationships (like mergesWith; see Section 5.9) and hates, where the column names must be disambiguated.

Multiple Entries. The table is only used for query transformation. Multiple entries mean that the referencing column has to be looked up in all *RangeTab*s, i.e., they have a disjunctive/union semantics.

4.4 Foreign Key Join Attributes of $n : m$ Tables

The NM Join Attributes (NMJ) table is a rather simple auxiliary table which serves mainly for quickly composing the join conditions (together with the RC table) when navigating through $n : m$ tables. It is a mapping

(Class ×) (N:M-)Table × LookupAttribute → FKJoinAttribute

with the semantics

> "For looking up the column *lookupattr* in table *nmtab* of an instance x of class *Class*, joining *nmtab* with the main table for *Class* via *nmtab.fkjoinattr* (i.e. the join condition is then *NMTab.fkjoinattr* = *myTable*.id), where *myTable* is the class table of x".

NMJoinAttrs			
Class	**NMTab**	**LookupAttr**	**FKJoinAttr**
:City	t_locOnIsland	Island	GeoThing
:Mountain	t_locOnIsland	Island	GeoThing
<unknown>	t_locOnIsland	Island	GeoThing
:Island	t_locOnIsland	GeoThing	Island
:City	t_locatedIn	Area	GeoThing
:Mountain	t_locatedIn	Area	GeoThing
:River	t_locatedIn	Area	GeoThing
<unknown>	t_locatedIn	Area	GeoThing
:Country	t_locatedIn-	GeoThing	Area
:Continent	t_locatedIn-	GeoThing	Area
<unknown>	t_locatedIn-	GeoThing	Area
:City	t_locatedAt	Water	City
:River	t_locatedAt	City	Water
:Lake	t_locatedAt	City	Water
:Sea	t_locatedAt	City	Water

The table can be created and filled by

```
CREATE TABLE NMJoinAttrs
AS
(SELECT md.Class, md.Table, c1.column_name, c2.column_name
 FROM mappingdict md, user_tab_cols c1, user_tab_cols c2
 WHERE md.Table IN (SELECT tablename from NMTables)
   AND md.Table = c1.table_name
   AND md.Table = c2.table_name
   AND md.LookupProp = c1.column_name
   AND c2.column_name <>  c1.column_name);  -- the other column
 -- TODO: handle WATER -> River/Lake/Sea etc
```

Note that even the *class* column is somewhat redundant, but makes things clearer (and safer). On the other hand, the table would be much smaller when removing it. But note that when misusing it for navigating through n-ary reified relationships as it will be discussed in Section 5.10, it is useful.

Example 14 *SPARQL queries are translated as follows using the medatata tables information:*

> ?C a :City; :name ?CN :locatedOnIsland ? I.
> ?I a :Island; :name ?IN .

```
SELECT c.name, i.name
FROM t_City c, t_locatedOnIsland l, t_Island i
WHERE l.GeoThing = c.id    -- from entry (NM4)
    AND l.Island = i.id;    -- from entry (RC8)
```

Multiple Entries. There are no multiple entries.

4.5 TOP Tables List

The table is a list, for which properties a separate TOP table exists. Since also the TOP table for functional properties has the structure of an $n:m$ table, the attribute names for composing the join condition are also given.

In the example scenario, there is so far no $n:m$ TOP property since the usual general $n:m$ tables cover all subclasses of the domain in the same table. TOP tables for $n:m$ properties are required for the refinements discussed in Sections 5.1, 5.3, 5.4, 5.5, and 5.6.

TopTables			
Property	**Table**	**LookupAttr**	**FKJoinAttr**
:name	TOP_name	name	oid
:population	TOP_population	population	oid
:area	TOP_area	area	oid
:capital	TOP_capital	capital	oid

The table can be created and filled by

```
CREATE TABLE TopTables
AS
(SELECT md1.Property, 'TOP_' || md1.Property, md1.LookupProp, 'oid'
 FROM mappingdict md1, mappingdict md2
 WHERE md1.Property = md2.Property
   AND md.Property is functional
   AND md1.Class < md2.Class)    -- create only once for each pair
```

4.6 Usage in the process

Loading Data: MD For loading data, only the metadata from the Mapping Dictionary is needed. The OC dictionary is filled on-the fly during loading whenever

- a triple of the form (s rdf:type c) is inserted, **or**
- a triple of the form (s p o) is inserted and the p is defined only for a single domain class (i.e., there is only one entry for the property p in the MD).
  [cf. Discussion in Section 5.8]

Note that the TOP tables list is not needed, but can rather be created during loading: whenever a triple (s p o) has to be stored, and

- s is not yet listed in the OC (i.e., its class is not yet stored), **and**

– p is defined at least for two domain classes (i.e., there is more than one entry for the property p in the MD),

then, $\mathsf{TOP}\_p(s, o)$ has to be added to the storage.

Query Transformation: All metadata tables The other metadata tables are only needed when transforming SPARQL queries to SQL (as illustrated by the above examples).

4.7 Alternative Meta Tables: Handling of $n : m$ Tables

The alternative concerns the representation of $n : m$ properties.

The above design treats $n : m$ properties as a symmetric modeling concept – as motivated from the ER-to-relational-transformation: $n : m$ tables are of the form $\mathsf{t\_p}(\mathsf{class}_1, \mathsf{class}_2)$. Thus, entries in the MD for using $n : m$ tables have always a "+" for the Inverse column – they are always interpreted as "forward": $\mathsf{class}_2.p^-$ has the MD entry $((\mathsf{class}_2, p^-)(t\_p, \mathsf{class}_1, +))$.

The alternative representation considers them as an extended, normalized version of class tables: instead of being symmetric, they have an id column, and a p column: $t\_p(id, p)$ (for people used to relational modeling, this is rather unintuitive). The semantics is to represent the multivalued property p of objects id where id acts as host object in the same was as in the class tables.

As a consequence, p^- cannot be seen in a symmetric way, but as inverse-navigation p^- through t_p to id. The MD entry is then $((\mathsf{class}_2, p^-)(t\_p, p, -))$. The syntactical transformation of queries is then analogous to the inverse of functional properties.

Using this modeling, the NMJ table is not necessary. Also, automatical query generation is simpler, since there is no case distinction between functional and multivalued properties. The resulting queries are the same (up to column renaming).

5 Discussion and Enhancements

The previous section dealt with the "obvious", straightforward transformation. In this section, some more insight and discussion is given for handling special situations.

Multi-Domain Properties Properties that are defined for several classes (in general, an abstract class or some DL class expression is given) are transformed as follows:

1. functional properties (like name, $\mathsf{population}$, or $\mathsf{capital}$ are distributed over the class tables. The inverse direction is encoded by multiple entries in the MD. Additionally, a TOP table is defined.
2. $n : m$ properties and non-functional literal-valued properties: the property has an own table that can incorporate all domain classes; there only the necessary entries in the MD, RC and NM metadata tables have to be made.

Thus, for case (2), a partitioning into non-abstract domain classes similar to (1) can be considered.

Polymorphic Properties: Multiple Ranges A similar discussion applies when the range consists of several classes.

The general case of partitioning multi-domain-multi-range properties into several tables is discussed in 5.3. This will show that this alternative alone has neither clear advantages, neither clear disadvantages.

Functional Subsets of Multi-Domain and/or Multi-Range Properties Clear advantages of a partitioning occur if it leads to the isolation of functional subsets of the property:

- locatedOnIsland is functional for mountains, but $n : m$ for cities.
- river.flowsInto is functional, but every river flows into (at most) one river, lake or sea.
- The same holds for the range of $n : m$ properties, e.g., again for locatedIn is an $n : m$ property, but it is functional for some combination of non-abstract domains and ranges, e.g. every city is located in exactly one country.

These topics are discussed in Sections 5.1, 5.4, and 5.5,

TOP tables for Single-Domain Functional Properties As discussed in Section 3.5, if a triple s p o where the class of s is not (yet) known, has to be inserted, a TOP table can be used. Alternatives for this topic is discussed in Section 5.8.

Recursive Non-Symmetric and Symmetric Properties A recursive property is one that correlated objects of some class with one or more objects of the *same* class. For non-functional, non-symmetric such properties only the naming of the $n : m$ table has to be fixed. For symmetric ones (that are in general non-functional), also storage aspects have to be discussed; this is discussed in Section 5.9.

5.1 Polymorphic (object-valued) Functional Properties

Consider the following relationships:

- A river (finally) flows into at most one water; which is either another river, a lake, or a sea:
  :flowsInto rdfs:domain :River; rdfs:range :Water; a owl:FunctionalProperty.
  (Note that actually, flowsInto is also defined (and also functional) for lakes to rivers; here the river issue is discussed isolatedly to develop its modeling; handling lakes simply adds the TOP issue and is described in Example 15).
- Another relationship flowsThrough states that a river flows *through* a lake. It is an $1 : n$ relationship (note that its inverse is functional since for a given lake, only one river can flow through it, other rivers flow then only into this lake).
  Note that there is a difference between flowing through a lake, and flowing only out of the lake.
  flowsThrough is *not* dealt with in this section.

The main goal is to add the property to the main table of the River class. There are two alternatives:

Single-Column, Multi-Reference Join: There is a single column River.flowsInto whose values refer to ids of rivers, lakes and seas:

MappingDict					
Class	**Property**	**Range**	**Table**	**LookupProp**	**Inv(+-)**
:River	:flowsInto	(:Water)	t_River	flowsInto	+
:Sea	:flowsInto-	(:Water)	t_River	flowsInto	-
:Lake	:flowsInto-	(:Water)	t_River	flowsInto	-
:River	:flowsInto-	(:Water)	t_River	flowsInto	-

RangeClassJoinAttrs		
Table	**Column**	**RangeTable**
t_River	flowsInto	t_River
t_River	flowsInto	t_Lake
t_River	flowsInto	t_Sea

Multiple entries denote that the range is given as an abstract class which is distributed over several class tables.

Separate Columns, Unique Reference-Join: There are three columns, dependent on the range class River.flowsInto_River, River.flowsInto_Lake, and River.flowsInto_Sea whose values refer to ids of either rivers, lakes or seas:

MappingDict					
Class	**Property**	**Range**	**Table**	**LookupProp**	**Inv(+-)**
:River	:flowsInto	:River	t_River	flowsInto_River	+
:River	:flowsInto	:Lake	t_River	flowsInto_Lake	+
:River	:flowsInto	:Sea	t_River	flowsInto_Sea	+
:Sea	:flowsInto-	(:River)	t_River	flowsInto_Sea	-
:Lake	:flowsInto-	(:River)	t_River	flowsInto_Lake	-
:River	:flowsInto-	(:River)	t_River	flowsInto_River	-

RangeClassJoinAttrs		
Table	**Column**	**RangeTable**
t_River	flowsInto_River	t_River
t_River	flowsInto_Lake	t_Lake
t_River	flowsInto_Sea	t_Sea

In this case, at least temporarily, a TOP table TOP_flowsInto(River,flowsInto) (or TOP_flowsInto(River,Water) or more general TOP_flowsInto(River,oid)) is required (for inserting triples (r flowsInto w) where the actual class of w is not yet known):

MappingDict					
Class	**Property**	**Range**	**Table**	**LookupProp**	**Inv(+-)**
:River	:flowsInto	(:Water)	TOP_flowsInto	Water	+
:Sea	:flowsInto-	(:Water)	TOP_flowsInto	River	+
:Lake	:flowsInto-	(:Water)	TOP_flowsInto	River	+
:River	:flowsInto-	(:Water)	TOP_flowsInto	River	+

RangeClassJoinAttrs		
Table	**Column**	**RangeTable**
TOP_flowsInto	River	t_River
TOP_flowsInto	Water	t_River
TOP_flowsInto	Water	t_Lake
TOP_flowsInto	Water	t_Sea

NMJoinAttrs			
Class	**NMTab**	**LookupAttr**	**FKJoinAttr**
River	TOP_flowsInto	Water	River
River	TOP_flowsInto	River	Water
Lake	TOP_flowsInto	River	Water
Sea	TOP_flowsInto	River	Water

The performance differences are not significant: Both cases require to join with the three target tables. The first solution saves some space, while the second one "looks" a bit faster, but by using indexes this is expected to be negligible.

Non-functional literal-valued Properties: The same considerations hold for non-functional literal-valued properties.

5.2 Polymorphic (object-valued) Multi-Domain Functional Properties

Polymorphic multi-domain functional properties combine the handling of multi-domain functional properties (as discussed already in Section 3 for name and capital) with the handling of polymorphic properties as shown in the previous section: (optionally) multiple columns and a TOP table.

Example 15 *Consider the generalization of flowsInto to lakes. A lake also flows into –usually– a river, but sometimes to another lake (Lake Superior, Lake Huron) or a sea (Lake Maracaibo, Caribbean Sea).*

Straightforwardly, as described for rivers in Section 5.1, this can be handled either by a single column flowsInto of the Lake table, or by three separated columns flowsInto_River, flowsInto_Lake, and flowsInto_Sea of the Lake table.

In the latter case, there is already a TOP_flowsInto table generated for temporarily storing rivers that flow into a water whose class is not yet known. This table also serves for tuples describing a water of unknown class (river or lake) that flows into some other water; only its column names have to be adapted to TOP_flowsInto(Water,flowsInto) or TOP_flowsInto(Water1,Water2).

In the former case, the TOP_flowsInto table is now also necessary.

MappingDict					
Class	***Property***	***Range***	***Table***	***LookupProp***	***Inv(+-)***
:Lake	*:flowsInto*	*:River*	*t_Lake*	*flowsInto_River*	*+*

:Lake	*:flowsInto*	*:Lake*	*t_Lake*	*flowsInto_Lake*	+
:Lake	*:flowsInto*	*:Sea*	*t_Lake*	*flowsInto_Sea*	+
:Sea	*:flowsInto-*	*:Lake*	*t_Lake*	*flowsInto_Sea*	-
:Lake	*:flowsInto-*	*:Lake*	*t_Lake*	*flowsInto_Lake*	-
:River	*:flowsInto-*	*:Lake*	*t_Lake*	*flowsInto_River*	-
:River	*:flowsInto*	*(:Water)*	*TOP_flowsInto*	*Water*	+
:Lake	*:flowsInto*	*(:Water)*	*TOP_flowsInto*	*flowsInto*	+
:Sea	*:flowsInto-*	*(:Water)*	*TOP_flowsInto*	*Water*	+
:Lake	*:flowsInto-*	*(:Water)*	*TOP_flowsInto*	*Water*	+
:River	*:flowsInto-*	*(:Water)*	*TOP_flowsInto*	*Water*	+

RangeClassJoinAttrs		
Table	***Column***	***RangeTable***
t_Lake	*flowsInto_River*	*t_River*
t_Lake	*flowsInto_Lake*	*t_Lake*
t_Lake	*flowsInto_Sea*	*t_Sea*
TOP_flowsInto	*Water*	*t_River*
TOP_flowsInto	*Water*	*t_Lake*
TOP_flowsInto	*Water*	*t_Sea*
TOP_flowsInto	*flowsInto*	*t_River*
TOP_flowsInto	*flowsInto*	*t_Lake*

NMJoinAttrs			
Class	***NMTab***	***LookupAttr***	***FKJoinAttr***
River	*TOP_flowsInto*	*flowsInto*	*Water*
Lake	*TOP_flowsInto*	*flowsInto*	*Water*
River	*TOP_flowsInto*	*Water*	*flowsInto*
Lake	*TOP_flowsInto*	*Water*	*flowsInto*
Sea	*TOP_flowsInto*	*Water*	*flowsInto*

5.3 General $n:m$ Multi-Domain-Multi-Range Properties

For an $n:m$ property p, the $n:m$ table $p(dom, rge)$ can be partitioned according to the non-abstract classes whose intersection with the domain/range is non-empty.

The most general case is a property $p: D \to R$ where the both domain D and range R are abstract (union) classes. With $D = D_1 \cup \ldots \cup D_k$ and $R = R_1 \cup \ldots \cup R_\ell$ being the non-abstract subclasses the property can be partitioned into $k \times \ell$ tables $D_i\_p\_R_j(D_i, R_j)$. In this case, also a TOP_p table is needed.

Example 16 (n:m relationship between abstract classes) *Consider the* locatedIn *relationship between all kinds of "small" geographical things, e.g., cities, mountains, rivers, lakes and all kinds of areas, e.g., provinces, countries, deserts, continents. The domain of* locatedIn *is an abstract, even unnamed, class (in Mondial defined as the intersection of* GeographicalThing *with the complement of* Continent*), and the range is the abstract class* area*.*

In the original mapping, the whole relationship is "simply" stored in a binary table t_locatedIn(GeoThing,Area)*. Both columns contain references to tuples in the class tables.*

It can be split in –among many others– tables City_locatedIn_Country(City,Country)*,* City_locatedIn_Province(City,Province)*,* City_locatedIn_Desert(City,Desert)*,* City_locatedIn_Continent(City,Conti Mountain_locatedIn_Province(Mountain,Province)*,* Mountain_locatedIn_Country(Mountain,Country)*,* River_locatedIn_Province(River,Province)*,* River_locatedIn_Country(River,Country) *etc. Some of the corresponding metadata table entries are given below:*

MappingDict					
Class	***Property***	***Range***	***Table***	***LookupProp***	***Inv(+-)***
City	*:locatedIn*	*:Province*	*City_locatedIn_Province*	*Province*	+
City	*:locatedIn*	*:Country*	*City_locatedIn_Country*	*Country*	+
City	*:locatedIn*	*:Continent*	*City_locatedIn_Continent*	*Continent*	+
City	*:locatedIn*	*<unknown>*	*TOP_locatedIn*	*Area*	+
Mountain	*:locatedIn*	*:Province*	*Mountain_locatedIn_Province*	*Province*	+
Mountain	*:locatedIn*	*:Country*	*Mountain_locatedIn_Country*	*Country*	+
Mountain	*:locatedIn*	*<unknown>*	*TOP_locatedIn*	*Area*	+
River	*:locatedIn*	*:Province*	*River_locatedIn_Province*	*Province*	+
River	*:locatedIn*	*:Country*	*River_locatedIn_Country*	*Country*	+
River	*:locatedIn*	*<unknown>*	*TOP_locatedIn*	*Area*	+
<unknown>	*:locatedIn*		*TOP_locatedIn*	*Area*	+
Province	*:locatedIn-*	*:City*	*City_locatedIn_Province*	*City*	+
Country	*:locatedIn-*	*:City*	*City_locatedIn_Country*	*City*	+
Province	*:locatedIn-*	*:Mountain*	*Mountain_locatedIn_Province*	*Mountain*	+
Country	*:locatedIn-*	*:Mountain*	*Mountain_locatedIn_Country*	*Mountain*	+

RangeClassJoinAttrs		
Table	***Column***	***RangeTable***
City_locatedIn_Province	*City*	*t_City*
City_locatedIn_Province	*Province*	*t_Province*
City_locatedIn_Country	*City*	*t_City*
City_locatedIn_Country	*Country*	*t_Country*
City_locatedIn_Continent	*Continent*	*t_Continent*
TOP_locatedIn	*Area*	*t_Province*
TOP_locatedIn	*Area*	*t_Country*
TOP_locatedIn	*Area*	*t_Continent*
TOP_locatedIn	*GeoThing*	*t_City*
TOP_locatedIn	*GeoThing*	*t_Mountain*
TOP_locatedIn	*GeoThing*	*t_River*

NMJoinAttrs			
Class	***NMTab***	***LookupAttr***	***FKJoinAttr***
:City	*City_locatedIn_Province*	*Province*	*City*

:Province	*City_locatedIn_Province*	*City*	*Province*
:City	*City_locatedIn_Country*	*Country*	*City*
:Country	*City_locatedIn_Country*	*City*	*Country*
:City	*Mountain_locatedIn_Country*	*Country*	*Mountain*
:Country	*Mountain_locatedIn_Country*	*Mountain*	*Country*
:City	*TOP_locatedIn*	*Area*	*GeoThing*
:Mountain	*TOP_locatedIn*	*Area*	*GeoThing*
:River	*TOP_locatedIn*	*Area*	*GeoThing*
:Province	*TOP_locatedIn*	*GeoThing*	*Area*
:Country	*TOP_locatedIn*	*GeoThing*	*Area*
:Continent	*TOP_locatedIn*	*GeoThing*	*Area*
<unknown>	*TOP_locatedIn*	*GeoThing*	*Area*

The space required for both alternatives is the same – since the same tuples are stored either in one table or partitioned over several tables. There are no multiple entries in the MD and in the NMJ table.
The performance differences are not significant:

- When the class-definite case is assumed, the TOP table is not needed after populating the database. Since there are no multiple entries in the MD and in the NMJ table, UNION queries are not necessary, also there are much less non-successful lookups for id joins. Nevertheless, using indexes, both is actually not expensive.
- in the non-class-definite case, the non-partitioned mapping is expected to be faster since only one (larger) join is required instead of several smaller ones; using indexes this is expected to be negligible.

5.4 $n : m$ Properties that are Functional on some Domain

A more interesting case for partitioning (by domain) are $n : m$ properties that are functional on some subset of their domain.

Example 17 *The property locatedOnIsland is only defined for cities and mountains: Cities can be located on (one or more) islands, while a mountain can only be located on a single island.*
The basic algorithm described in Section 3.1 thus defines a general $n : m$ table t_locatedOnIsland(oid,Island) with Mapping Dictionary entries

MappingDict					
Class	***Property***	***Range***	***Table***	***LookupProp***	***Inv(+-)***
:City	*:locOnIsland*	*(:Island)*	*t_locOnIsland*	*Island*	+
:Mountain	*:locOnIsland*	*(:Island)*	*t_locOnIsland*	*Island*	+
<unknown>	*:locOnIsland*	*(:Island)*	*t_locOnIsland*	*Island*	+
:Island	*:locOnIsland-*	*(:GeoThing)*	*t_locOnIsland*	*GeoThing*	+

Partitioning it results in a table City_locOnIsland(City,Island), and the functional property Mountain.locOnIsland is added to the Mountain class table which is then t_Mountain(oid,name,elevation,...,locOnIsland).

Note that then, also a TOP table TOP_locatedOnIsland(oid,Island) *is needed since upon insert of a tuple* $(s$ locatedOnIsland $i)$ *without further information, even in the class-definite case, it is not known what class* s *belongs to. The pair* (s, i) *is inserted into* TOP_locatedOnIsland(oid,Island) *until class information about* s *becomes available.*

The metadata entries are as follows:

MappingDict					
Class	***Property***	***Range***	***Table***	***LookupProp***	***Inv(+-)***
:City	*:locOnIsland*	*(:Island)*	*City_locOnIsland*	*Island*	*+*
:Mountain	*:locOnIsland*	*(:Island)*	*t_Mountain*	*locOnIsland*	*+*
<unknown>	*:locOnIsland*	*(:Island)*	*TOP_locOnIsland*	*Island*	*+*
:Island	*:locOnIsland-*	*:City*	*t_City_locOnIsland*	*City*	*+*
:Island	*:locOnIsland-*	*:Mountain*	*t_Mountain*	*locOnIsland*	*-*
:Island	*:locOnIsland-*	*<unknown>*	*TOP_locOnIsland*	*GeoThing*	*+*

RangeClassJoinAttrs		
Table	***Column***	***RangeTable***
Mountain	*locOnIsland*	*t_Island*
City_locOnIsland	*City*	*t_City*
City_locOnIsland	*Island*	*t_Island*
TOP_locOnIsland	*GeoThing*	*t_City*
TOP_locOnIsland	*GeoThing*	*t_Mountain*
TOP_locOnIsland	*Island*	*t_Island*

NMJoinAttrs			
Class	***NMTab***	***LookupAttr***	***FKJoinAttr***
:City	*City_locOnIsland*	*Island*	*City*
:Island	*City_locOnIsland*	*City*	*Island*
<unknown>	*TOP_locOnIsland*	*Island*	*GeoThing*
:Island	*TOP_locOnIsland*	*GeoThing*	*Island*

Consider the queries

```
?C a :City; :name ?CN; :locOnIsland [:name ?N].
?M a :Mountain; :name ?MN; :locOnIsland [:name ?N].
```

The MD and NMJ tables state that City.locOnIsland *is looked up in* City_locOnIsland.Island *by joining via* t_City.id = City_locOnIsland.City *and then (RC) joining* City_locOnIsland.Island = t_Island.id.

On the other hand, Mountain.locOnIsland *is looked up in* t_Mountain.locOnIsland *and then (RC) joining* t_Mountain.locOnIsland = t_Island.id:

```
SELECT c.name, i.name
FROM t_City c, City_locOnIsland cl, t_Island i
WHERE c.id = cl.City
    AND cl.Island = i.id;
```

```
SELECT m.name, i.name
FROM t_Mountain m, t_Island i
WHERE m.locOnIsland = i.id;
```

As the example shows, the query for the functional property needs one join less. Thus, if a subset of a $n:m$ property is functional, partitioning is strongly recommended.

5.5 Polymorphic (object-valued) $n:m$ Properties

Consider the following relationship:

– Cities can be located at (locatedAt) waters, i.e., rivers, lakes, seas. A city can be located at several instances of each of these subclasses.

The domain of the relationship is a non-abstract class (City), and the range is an abstract class (Water). Similar to the (functional and exclusive) flowsInto case in Section 5.1, there are different possibilities.

Single-Table, Multi-Reference Join: The basic algorithm described in Section 3.1 creates a single table t_locatedAt(City, Water) whose first column refers to a city, and the second column refers to ids of rivers, lakes and seas:

MappingDict					
Class	**Property**	**Range**	**Table**	**LookupProp**	**Inv(+-)**
:City	:locatedAt	(:Water)	t_locatedAt	Water	+
:Sea	:locatedAt-	(:City)	t_locatedAt	City	+
:Lake	:locatedAt-	(:City)	t_locatedAt	City	+
:River	:locatedAt-	(:City)	t_locatedAt	City	+

RangeClassJoinAttrs		
Table	**Column**	**RangeTable**
t_locatedAt	City	t_City
t_locatedAt	Water	t_River
t_locatedAt	Water	t_Lake
t_locatedAt	Water	t_Sea

Multiple entries (as for t_locatedAt.Water) in the RC table denote that the range is given as an abstract class which is distributed over several class tables.

NMJoinAttrs			
Class	**NMTab**	**LookupAttr**	**FKJoinAttr**
:City	t_locatedAt	Water	City
:River	t_locatedAt	City	Water
:Lake	t_locatedAt	City	Water

:Sea	t_locatedAt	City	Water

Separate Tables, Unique Reference-Join: Partitioning wrt. the non-abstract subclasses of the range yields three tables, one for each range class locatedAt_River(City,River), locatedAt_Lake(City,Lake), locatedAt_Sea(City,Sea), whose values refer to ids of either rivers, lakes or seas.

MappingDict					
Class	**Property**	**Range**	**Table**	**LookupProp**	**Inv(+-)**
:City	:locatedAt	:River	locatedAt_River	River	+
:City	:locatedAt	:Lake	locatedAt_Lake	Lake	+
:City	:locatedAt	:Sea	locatedAt_Sea	Sea	+
:Sea	:locatedAt-	(:City)	locatedAt_River	City	-
:Lake	:locatedAt-	(:City)	locatedAt_Lake	City	-
:River	:locatedAt-	(:City)	locatedAt_Sea	City	-

RangeClassJoinAttrs		
Table	**Column**	**RangeTable**
locatedAt_River	City	t_City
locatedAt_Lake	City	t_City
locatedAt_Sea	City	t_City
locatedAt_River	River	t_River
locatedAt_Lake	Lake	t_Lake
locatedAt_Sea	Sea	t_Sea

NMJoinAttrs			
Class	**NMTab**	**LookupAttr**	**FKJoinAttr**
:City	locatedAt_River	River	City
:City	locatedAt_Lake	Lake	City
:City	locatedAt_Sea	Sea	City
:River	locatedAt_River	City	River
:Lake	locatedAt_Lake	City	Lake
:Sea	locatedAt_Sea	City	Sea

In this case, at least temporarily a TOP table TOP_locatedAt(City,Water) (or more general TOP_locatedAt(City,oid)) is required (for inserting triples (c locatedAt w) where the actual class of w is not yet known:

MappingDict					
Class	**Property**	**Range**	**Table**	**LookupProp**	**Inv(+-)**
:City	:locatedAt	(:Water)	TOP_locatedAt	Water	+
:Sea	:locatedAt-	(:City)	TOP_locatedAt	City	-
:Lake	:locatedAt-	(:City)	TOP_locatedAt	City	-
:River	:locatedAt-	(:City)	TOP_locatedAt	City	-

RangeClassJoinAttrs		
Table	**Column**	**RangeTable**
TOP_locatedAt	City	t_City
TOP_locatedAt	Water	t_River
TOP_locatedAt	Water	t_Lake
TOP_locatedAt	Water	t_Sea

NMJoinAttrs			
Class	**NMTab**	**LookupAttr**	**FKJoinAttr**
:City	TOP_locatedAt	Water	City
:River	TOP_locatedAt	City	Water
:Lake	TOP_locatedAt	City	Water
:Sea	TOP_locatedAt	City	Water

The performance is not expected to change much, as described in Section 5.3.

5.6 $n:m$ Properties with Functional Subsets

A more interesting case is the following:

- The locatedIn property (that was already dealt with as the general case of $n:m$ with multiple domain and range subclasses) is defined in a very general way.
  The most important (and in Mondial actually stored) relationships are between the above objects and provinces/countries (to have a consistent naming, for seas, locatedIn denotes their relationship with the bordering countries and provinces).
- The relevant restriction here is that cities are located exactly in one country and at most in one province (in case that in a country, no provinces are stored). Thus, the subsets locatedIn: City → Country and locatedIn: City → Province are functional; and these are frequently used for joins[4].
  Analogously, locatedIn: Province → Country is <1,1>, i.e., also functional.

The latter characteristics can be used to separate the functional subsets locatedIn: City → Country, locatedIn: City → Province, and locatedIn: Province → Province not into $n:m$ tables City_locatedIn_Country, City_locatedIn_Province, and Province_locatedIn_Country, but to incorporate them into the main tables for cities and provinces as t_City(id, name, locIn_Country, locIn_province, population,...) and t_Province(id, name, locIn_country, population,...).

For the remaining tuples, the same considerations as in Section 5.3 hold:

- $k \times \ell$ tables $D_i\_p\_R_j$ and a TOP_p table, or

[4] in the relational model, (name, country, province) form the key of the city table; for countries where no provinces are stored, the country itself is considered to be the only province)

– only a large t_p table (for all domains and ranges, which also acts as TOP table). The latter is recommended if k and ℓ are large which would lead to a large number of different tables.

Below, the metadata tuples when a general t_locatedIn(GeoThing,Area) table which collects all non-functional subsets is used are given below.

MappingDict					
Class	**Property**	**Range**	**Table**	**LookupProp**	**Inv(+-)**
:City	:locatedIn	:Country	t_City	locIn_Country	+
:City	:locatedIn	:Province	t_City	locIn_Province	+
:Province	:locatedIn	:Country	t_Province	locIn_Country	+
:City	:locatedIn		t_locatedIn	Area	+
:Mountain	:locatedIn		t_locatedIn	Area	+
:River	:locatedIn		t_locatedIn	Area	+
:Sea	:locatedIn		t_locatedIn	Area	+
:Province	:locatedIn-	:City	t_City	locIn_Province	-
:Country	:locatedIn-	:City	t_City	locIn_Country	-
:Country	:locatedIn-	:Province	t_Province	locIn_Country	-
:Province	:locatedIn-	<unknown>	t_locatedIn	GeoThing	+
:Country	:locatedIn-	<unknown>	t_locatedIn	GeoThing	+
:Continent	:locatedIn-	<unknown>	t_locatedIn	GeoThing	+

Note that for locatedIn of cities wrt. areas that are not provinces or countries (e.g., islands, deserts, continents), these tuples are still stored in the general t_locatedIn table. This aspect will be discussed from an operational point of view below.

RangeClassJoinAttrs		
Table	**Column**	**RangeTable**
t_City	locIn_Country	t_Country
t_City	locIn_Province	t_Province
t_Province	locIn_Country	t_Country
t_locatedIn	GeoThing	t_City
t_locatedIn	GeoThing	t_Province
t_locatedIn	GeoThing	t_Country
t_locatedIn	GeoThing	t_Mountain
t_locatedIn	GeoThing	t_River
t_locatedIn	GeoThing	t_Sea
t_locatedIn	Area	t_Province
t_locatedIn	Area	t_Country
t_locatedIn	Area	t_Continent
t_locatedIn	Area	t_Desert
t_locatedIn	Area	t_Sea

NMJoinAttrs			
Class	**NMTab**	**LookupAttr**	**FKJoinAttr**
:City	t_locatedIn	Area	GeoThing
:Province	t_locatedIn	Area	GeoThing
:Country	t_locatedIn	Area	GeoThing
:Mountain	t_locatedIn	Area	GeoThing
:River	t_locatedIn	Area	GeoThing

:Lake	t_locatedIn	Area	GeoThing
:Province	t_locatedIn	GeoThing	Area
:Country	t_locatedIn	GeoThing	Area
:Continent	t_locatedIn	GeoThing	Area
:Sea	t_locatedIn	GeoThing	Area
:Desert	t_locatedIn	GeoThing	Area

This actually saves to compute one join for queries of the form

```
?X a :City; :name ?XN; :locatedIn ?Y.
?Y a :Country; : name ?YN.
```

which then only requires t_City $\bowtie$ t_Country instead of t_City $\bowtie$ t_locatedIn $\bowtie$ t_Country (assuming the class-definite case).

Operational Aspects. For query evaluation, only the separated functional property has to be evaluated. Consider again the relevant entries in the Mapping Dictionary:

MappingDict					
Class	**Property**	**Range**	**Table**	**LookupProp**	**Inv(+-)**
:City	:locatedIn	:Country	t_City	locIn_Country	+
:City	:locatedIn	:Province	t_City	locIn_Province	+
:City	:locatedIn	<unknown>	t_locatedIn	Area	+

Note that a similar situation occured for polymorphic (functional) properties in Section 5.1, but with a different semantics: there, in the class-definite case, the contents of the general column was completely partitioned over the separated columns (and the TOP table is finally empty or even removed); in the non-class-definite case, the semantics of the Mapping Dictionary is disjunctive – to look up the property both in the respective separated columns and in the TOP table.

Here, the separated columns are to be looked up *instead* of the general table if th domain is City or Province and the range is Province or Country – but to look up in the general table, if the domain is any other GeoObject, or if the range is Island, Desert etc.

5.7 $n : m$ Properties with Functional and Inverse Functional Subsets

Note that in the same way, also $n : m$ properties r that have some functional subsets, and also some inverse functional subsets can be partitioned:

- partition first into into subsets $D_i \times R_j$
- functional ones are represented in the class table of D_i, with MD entries $((D_i, r, R_j)(D_i, r\_R_j, +))$ and $((R_j, r^-, D_i)(D_i, r\_R_j, -))$;
- inverse functional ones are represented in the class table of R_j, with MD entries $((R_j, r^-, D_i)(D_i, r\_R_j, +))$ and $((D_i, r, R_j)(D_i, r\_R_j, -))$;
- non-functional and non-inverse-functional ones are represented either

- each subset in an $n:m$ table $D_i\_p\_R_j(D_i, R_j)$ as described for locatedIn in Example 16 with MD entries $((D_i, r, R_j)(D_i\_r\_R_j, R_j, +))$ and $((R_j, r^-, D_i)(D_i\_r\_R_j, D_i, +))$ (then, a TOP table is at teast temporarily needed), or
- all together in the "general" table t_r as described for locatedIn in Section 5.6. with entries $((D_i, r)(t\_r, R, +))$ and $((R_j, r^-)(t\_r, D, +))$ (then, t_r serves also as TOP table).

5.8 TOP Tables: Alternatives

As discussed in Section 3.5, if a triple s p o where the class of s is not (yet) known has to be inserted, a TOP table can be used. In the plain transformation discussed in Section 3, this applied only to functional properties (that are modeled in the class tables, each $n:m$ relationship is represented in a single table); with the above refinements for $n:m$ relationships by partitioning into domain classes, the same applies also for them.

- if the domain of p is only a single class c, the class c of s can be derived (and (s, c) can be stored in the OC) and the triple can immediately be stored in the respective table.
  Thus TOP tables are in the basic modeling only *necessary* for functional properties whose domain overlaps with at least two classes.
- A "lazy" processing stores all triples first in TOP tables, and the final tables are only filled by SQL bulk processing when finally the class membership of all objects is known. This obviously requires to create TOP tables for all properties.
- several intermediate forms are possible.

5.9 Symmetric Recursive (object-valued) Non-Functional Properties

Recursive properties are properties with domain(r)=range(r); the naming conventions have already been described in the algortithm given in Section 3.1.

Symmetric recursive $n:m$ relationships can be stored only one-direction ("antisymmetric"), or (redundantly) both directions, which makes query formulation easier (no "union all") and slightly faster. Also, less entries in the above tables are needed.

Antisymmetric With antisymmetric storage, only one direction of each instance of the relationship is stored explicitly (the choice is arbitrary, often one uses e.g. $x_1 < x_2$ for $r(x, y)$. For correct evaluation, both directions must be listed in the MD (recall that the semantics of multiple entries is disjunctive) to be used:

MappingDict

Class	Property	Range	Table	LookupProp	Inv(+-)
:Sea	:mergesWith	NO ENTRY!	t_mergesWith	Sea2	+
:Sea	:mergesWith	NO ENTRY!	t_mergesWith	Sea1	+
:Sea	:mergesWith-	NO ENTRY!	t_mergesWith	Sea2	+
:Sea	:mergesWith-	NO ENTRY!	t_mergesWith	Sea1	+

means that both entries must be looked up (note that the range column must not be filled because it would mark the entry to be exclusive, cf. Section 5.6). Note that the MD is also used for inserting triples – thus multiple entries make the insertion ambiguous, which is exactly the semantics of arbitrarily choosing one direction to store.

RangeClassJoinAttrs		
Table	**Column**	**RangeTable**
t_mergesWith	Sea1	t_Sea
t_mergesWith	Sea2	t_Sea

NMJoinAttrs			
Class	**NMTab**	**LookupAttr**	**FKJoinAttr**
:Sea	t_mergesWith	Sea2	Sea1
:Sea	t_mergesWith	Sea1	Sea2

Symmetric/Redundant In the symmetric case, one direction is considered to be r, the other is r^-:

MappingDict					
Class	**Property**	**Range**	**Table**	**LookupProp**	**Inv(+-)**
:Sea	:mergesWith	(:Sea)	t_mergesWith	Sea2	+
:Sea	:mergesWith-	(:Sea)	t_mergesWith	Sea1	+

The RC and NMJ entries are the same as in the nonsymmetric case.

Note that it is easy to define a symmetric view based on the antisymmetric storage.

The handling of symmetric properties that are functional and/or inverse-functional (like e.g., "marriedWith") is integrated within the class tables where the storage is inherently symmetric and redundant. Here, an antisymmetric storage would be unnatural.

5.10 Making Reified Properties Accessible as Properties

Reified properties are modeled (as usual in the relational model) by artificial (reified) entity types and their tables. The respective tables have two (for binary relationships; n foreign keys for n-ary relationships) foreign keys that reference the involved instances of the entity types, and additionally columns for the attributes (if applicable). By this, they can be seen as generalizations of the binary $n:m$ tables.

The modeling described in Section 3.1 shows this even more: the $1:n$-relationships between a reified entity type and the involved "true" entity types

are deleted, and only the reified table remains with its foreign keys – just in the same place as a "normal" $n:m$ table.

This similarity can be used to add the reified relationship as binary $n:m$ relationship to the usable property names in a very simple way: let R be the entity type that reifies a relationship r between entity types C and D with attributes $a_1, \ldots, a_n$. Let the auxiliary relationships between C and R and D and R be cr and dr:

- The resulting "class" table of R is $t\_R(c, d, a_1, \ldots, a_n)$. The MD entries are $((C, cr^-)(t\_R, c, -))$, $((D, dr^-)(t\_R, d, -))$, $((R, cr)(t\_R, c, +))$, $((R, dr)(t\_R, d, +))$, and $((R, a_i)(t\_R, a_i, +)$.
- The RC entries are (R, c, t_C) and (R, d, t_D).
- There are no NMJ entries since t_R is not an $n:m$ table, but a class table for the reified class R.

The projection $\pi[c, d](t\_R)$ is actually an $n:m$ table between C and D – namely the $n:m$ table of the relationship type r that was reified into R.
With appropriate entries in the metadata tables, t_R can be used as the $n:m$ table for r:

- MD entries $((C, r)(t\_R, d, +))$ and $((D, r^-)(t\_R, c, +))$.
- There are no new/different RC entries (the foreign keys of t_R stay the same).
- t_R acts as $n:m$ table with NM table entries (C, t_R, d, c) and (D, t_R, c, d).

Example 18 *Consider again Example* **??***: the* encompassed*-relationship is reified into the entity type* EncompBy *whose MD and RC entries have been given in Section 4. The* encompassed *relationship, implemented by interpreting the π[countryEncBy-,encCont](t_EncompBy) as $n:m$ table is made accessible as follows:*

MappingDict					
Class	***Property***	***Range***	***Table***	***LookupProp***	***Inv(+-)***
:Country	*:encompassed*	*(:Continent)*	*t_EncompBy*	*encCont*	-
:Continent	*:encompassed-*	*(:Country)*	*t_EncompBy*	*countryEncBy-*	-

NMJoinAttrs			
Class	***NMTab***	***LookupAttr***	***FKJoinAttr***
:Country	*t_EncompBy*	*encCont*	*countryEncBy-*
:Continent	*t_EncompBy*	*countryEncBy-*	*encCont*

Then, queries using encompassed *can seamlessly be transformed*

```
?C a :Country; :name ?CN; :encompassed [ :name ?Cont].
```

```
SELECT c.name, co.name
FROM t_Country c, t_EncompBy e, t_Continent co
WHERE c.id = e.countryEncBy-
    AND e.encCont = co.id;
```

Note the similarity of the SQL statement with the following query that additionally returns the percentage:

```
?C a :Country; :countryEncBy ?E.
?E a :EncompBy; :percent ?P; :encCont ?CO
?CO a :Continent; :name ?CN.
```

```
SELECT c.name, e.percent, co.name
FROM t_Country c, t_EncompBy e, t_Continent co
WHERE c.id = e.countryEncBy-
    AND e.encCont = co.id;
```

The tuples for the NMJ table show that the SQL statement given in Section 4.4 to create the NMJ table as a *view* over the schema definition does not cover this case: "take the other column" is ambiguous, since in this "$n : m$+attributes" table there are two "other" columns. Nevertheless, the RC table would be sufficient for determining the "other" reference attribute.

Note that in `mondial-meta.n3`, the reifies (meta-)property states which Properties are reified by some ReifiedRelationship. The appropriate MD and NMJ entries can be generated automatically.

6 Subclasses and Subproperties

6.1 Pure RDF vs RDFS vs OWL

Subclass and Subproperty relationships do not belong to *plain* RDF. In RDF there are simply data triples. From that point of view, only the *metadata* ontology uses RDFS and OWL from which the Mapping Dictionary can be derived. The actual input data consists only of triples, and the queries are evaluated only wrt. these explicit triples.

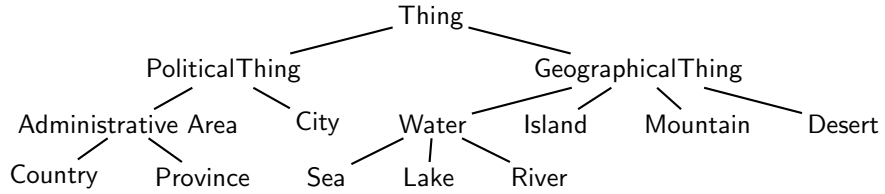
With RDFS, simple subclass and subproperty statements for "upward" closure, can easily be expressed. Downward definitions, when an object must be an instance of a finer class can only be done in OWL using standard and rather easy-to-understand DL notions. Downward definitions when a relationship must be an instance of a finer property is only possible even in OWL (to the best of our knowledge) when the maximal cardinality is 1. (note that with rules, this is can be expressed in a very easy way).

The following sections analyze (A) the handling of subclass issues, and (B) the handling of subproperty issues.

6.2 A: Subclasses/Abstract Modeling Classes

Any support of subclassing like rdfs:subClassOf requires reasoning capabilities.

Possible: Reasoning with a simple class hierarchy with statements sc rdfs:subClassOf c.
In this case, the RDF input data must contain triples o rdf:type x for the *most specific* class/classes where o belongs to.
This can relatively easily be implemented:
If a class hierarchy is modeled, e.g.



or even a class network with multiple inheritance where Seas and Administrative Areas are also Areas, there is always the question how to model these classes. There is no standard suggestion, but several alternatives (amongst which some are only conditionally applicable):

- Having only tables for leaf classes. This is only possible, if the leaves *cover* their subclasses, e.g., every water to be considered is a lake, a river, or a sea.
  The tables then contain all properties of the respective superclasses.
- Having tables for leaf classes and for non-leaf classes: On every path from Thing to any leaf class, (at least) one of the classes must be represented by a table. This table must contain all properties of its superclasses, and also the union of all properties of its subclasses. Properties that are not common to all leaf classes will be filled with null values.
  There is an advantage of this modeling if subclasses are not disjoint: each common instance is represented only by a single tuple.
- Having tables for each class. Every table has exactly the properties that are modeled within the respective class.
  The data about instances is thus spread over several tables. This is a disadvantage if several properties are needed together. On the other hand, it allows efficient union-free access on all objects of a non-leaf class with their common properties.
- the above alternatives may be mixed.
- SQL views, or even redundant materialized storage may also be applied.

Each of the above storage principles can easily be represented by appropriate entries in the Mapping Dictionary and the other metadata tables.

Example 19 *Consider the Mondial example extended with non-covering superclasses: There are waters that are neither rivers, nor lakes, nor seas, e.g., canals (note that this can be determined from the RDFS/OWL schema, whether* $\mathsf{Water} \equiv \mathsf{River} \sqcup \mathsf{Lake} \sqcup \mathsf{Sea}$ *(covering) or* $\mathsf{Water} \sqsupseteq \mathsf{River} \sqcup \mathsf{Lake} \sqcup \mathsf{Sea}$ *(potentially non-covering))*
In this case, there must be a class table t_Water(id,name)*, and such waters are also allowed in all properties where a* Water *or a* GeoThing *is allowed, e.g., cities can be located at it, and waters are located in areas.*

MappingDict					
Class	***Property***	***Range***	***Table***	***LookupProp***	***Inv(+-)***
:Water	*:name*		*t_Water*	*name*	*+*

:Water	*:name*	*t_River*	*name*	+
:Water	*:name*	*t_Lake*	*name*	+
:Water	*:name*	*t_Sea*	*name*	+
:Water	*:locatedIn*	*TOP_locatedIn*	*Area*	+

RangeClassJoinAttrs		
Table	***Column***	***RangeTable***
TOP_flowsInto	*Water*	*t_Water*
TOP_locatedAt	*Water*	*t_Water*
TOP_locatedIn	*GeoThing*	*t_Water*

NMJoinAttrs			
Class	***NMTab***	***LookupAttr***	***FKJoinAttr***
:Water	*t_locatedIn*	*Area*	*GeoThing*

Note that there are now four MD entries for (:Water,:name) because the name of a water can be found/stored in the Water, River, Lake, or Sea table. For querying, all tables have been joined/looked up by a UNION query.

For storing triples about subjects that are only known to be waters, the table entries are ambiguous. Obviously, they must be stored in the t_Water table (which in some way is a lower proxy for the TOP_name table). (note that a similar situation was before handled by using <unknown> as class, that gave a hint to store in TOP.)

The situation is solved by

- *use the alternative whose table name corresponds to the class name, or*
- *explicitly adding a W/R (Write – unambiguous, and Read - disjunctive/union) annotation as additional column.*

Note that information about cities located at such a water is put into the t_locatedAt table if locatedAt is stored in a single $n : m$ table, while it is put into the TOP_locatedAt table if locatedAt is stored in separate locatedAt_Watertype $n : m$ tables. In this case, similar to the non-class-definite case, the TOP table is not finally empty.

The optimal solution cannot be derived from the schema, but has to be based on data statistics and query statistics. Note that with the Mapping Dictionary that treats each property separately, our approach allows also for a dynamic restructuring at runtime.

Impossible: the other way round, *defining* a subclass as the set of all elements of a superclass that satisfy a certain condition is possible in OWL via owl:Restriction.

In this case, given a RDF triple o rdf:type c where actually o is also an instance of a subclass sc of c, the latter must be *derived* by OWL reasoning. This is not acceptable for the intended application.

6.3 B: Subproperties

In contrast to the class hierarchy, the more general properties are often not abstract but simple more general (as a proper superset) than the subproperties.

Example 20 *In the case*

:hasSon rdfs:subPropertyOf :hasChild.
:hasDaughter rdfs:subPropertyOf :hasChild.

the superproperty is covering and abstract, and the case can be solved similar to the above class hierarchy: the relationships hasSon hasDaughter *are* $n : m$ *(actually,* $2 : m$*), requiring* $n : m$ *tables* hasSon(Person,hasSon) *and* hasSon(Person,hasDaughter) *(named by the pattern* $prop(c, prop)$*).*

MappingDict					
Class	***Property***	***Range***	***Table***	***LookupProp***	***Inv(+-)***
:Person	*:hasSon*		*t_hasSon*	*hasSon*	+
:Person	*:hasSon-*		*t_hasSon*	*Person*	+
:Person	*:hasDaughter*		*t_hasDaughter*	*hasDaughter*	+
:Person	*:hasDaughter-*		*t_hasDaughter*	*Person*	+
:Person	*:hasChild*		*t_hasSon*	*hasSon*	+
:Person	*:hasChild*		*t_hasDaughter*	*hasDaughter*	+
:Person	*:hasChild-*		*t_hasSon*	*Person*	+
:Person	*:hasChild-*		*t_hasDaughter*	*Person*	+

*Note that the definition the other way round "*hasSon *is the set of all instances of* hasChild *s.t. the child is male" is not possible in general in RDFS and/or OWL. Below, it is shown that such a definition is possible on OWL if the cardinality is either fixed or* $< 0, 1 >$*.*

In the following section, it is shown that such a definition is possible in our approach if the range set that defines the subproperty is a (non-abstract) class.

Consider another example from the Mondial application:

Example 21 *Consider again the –general– relationship* locatedIn(Place, Area)*, and the subproperties* inCountry(City,Country)*,* inProvince(City,Province)*, and* belongsTo(Province,Country)*. In addition to just being subproperties, the* definition *holds that their extensions are* maximal *wrt.* locatedIn*, and that the following cardinality restrictions hold:*

- *every city is located in* exactly *one* country*:* $City \subseteq \exists\ 1: inCountry.T$.
- *every city is located in* at most *one* province *(if in a country, no provinces are known, it is not located in a province):* $City \subseteq \exists \leq 1: inProvince.T$.
- *every province is located in* exactly *one* country*:* $Province \subseteq \exists\ 1: belongsTo.T$.

The above cardinality constraints (but not the definitions) can be expressed in OWL (via owl:Restriction*) in an obvious way, while the definitions of subproperties can only be expressed via awkward OWL axioms (cf. Section* **??***).*

The separation of the above functional subsets of locatedIn *has already been discussed in Section 5.6. The "definitions" of* inCountry*,* inProvince*, and* belongsTo *can simply be done by appropriate entries in the Mapping Dictionary:*

MappingDict					
Class	***Property***	***Range***	***Table***	***LookupProp***	***Inv(+-)***
:City	*:inCountry*	*(:Country)*	*t_City*	*locIn_Country*	*+*
:City	*:inProvince*	*(:Province)*	*t_City*	*locIn_Province*	*+*
:Province	*:belongsTo*	*(:Country)*	*t_Province*	*locIn_Country*	*+*
:Country	*:inCountry-*	*:City*	*t_City*	*locIn_Country*	*-*
:Country	*:inCountry-*	*:Province*	*t_Province*	*locIn_Country*	*-*
:Country	*:belongsTo-*	*(:Province)*	*t_Province*	*locIn_Country*	*-*

Note that there are no additional entries in the RC and NMJ tables.

Example 22 *The above example can be extended to non-functional subproperties defined in the same way. Assume one wants to define* inCountry *also for mountains as the restriction of* locatedIn *to* Mountain × Country *(which is, as shown in Section 6.3, not possible in OWL).*

MappingDict					
Class	***Property***	***Range***	***Table***	***LookupProp***	***Inv(+-)***
:Mountain	*:inCountry*		*TOP_locatedIn*	*Area*	*+*
:Country	*:inCountry-*	*Mountain*	*TOP_locatedIn*	*GeoThing*	*+*

Note that for the restriction of the inverse to mountains, there must be some more information. Otherwise, all rivers, lakes etc. that are stored in the same table would also be returned. Partitioning of TOP_locatedIn *by domain (resulting in, among others, a table* Mountain_locatedIn*) would do it.*

7 Complete Example Tables

MappingDict					
Class	**Property**	**Range**	**Table**	**LookupProp**	**Inv(+-)**
:City	:name		t_City	name	+
:Country	:name		t_Country	name	+
:Continent	:name		t_Country	name	+
<unknown>	:name		TOP_name	name	+
:City	:population		t_City	population	+
:Country	:population		t_Country	population	+
<unknown>	:population		TOP_population	population	+
:Country	:area		t_Country	area	+
:Continent	:area		t_Continent	area	+
<unknown>	:area		TOP_area	area	+
:Country	:capital	(:City)	t_Country	capital	+
:Province	:capital	(:City)	t_Province	capital	+
<unknown>	:capital	(:City)	TOP_capital	capital	+

:City	:capital-	:Country	t_Country	capital	-
:City	:capital-	:Province	t_Province	capital	-
:City	:capital-	<unknown>	TOP_capital	capital	-
:EncompBy	:countryEncBy-	(:Country)	t_EncompBy	countryEncBy$^-$	+
:EncompBy	:encCont	(:Continent)	t_EncompBy	encCont	+
:Country	:countryEncBy	(:EncompBy)	t_EncompBy	countryEncBy$^-$	-
:Continent	:encCont-	(:EncompBy)	t_EncompBy	encCont	-
:River	:flowsInto	:River	t_River	flowsInto_River	+
:River	:flowsInto	:Lake	t_River	flowsInto_Lake	+
:River	:flowsInto	:Sea	t_River	flowsInto_Sea	+
:Sea	:flowsInto-	(:River)	t_River	flowsInto_Sea	-
:Lake	:flowsInto-	(:River)	t_River	flowsInto_Lake	-
:River	:flowsInto-	(:River)	t_River	flowsInto_River	-
:Lake	:flowsInto	:River	t_Lake	flowsInto_River	+
:Lake	:flowsInto	:Lake	t_Lake	flowsInto_Lake	+
:Lake	:flowsInto	:Sea	t_Lake	flowsInto_Sea	+
:Sea	:flowsInto-	:Lake	t_Lake	flowsInto_Sea	-
:Lake	:flowsInto-	:Lake	t_Lake	flowsInto_Lake	-
:River	:flowsInto-	:Lake	t_Lake	flowsInto_River	-
:River	:flowsInto	(:Water)	TOP_flowsInto	Water	+
:Lake	:flowsInto	(:Water)	TOP_flowsInto	flowsInto	+
:Sea	:flowsInto-	(:Water)	TOP_flowsInto	Water	+
:Lake	:flowsInto-	(:Water)	TOP_flowsInto	Water	+
:River	:flowsInto-	(:Water)	TOP_flowsInto	Water	+
:City	:locOnIsland	(:Island)	City_locOnIsland	Island	+
:Mountain	:locOnIsland	(:Island)	t_Mountain	locOnIsland	+
<unknown>	:locOnIsland	(:Island)	TOP_locOnIsland	Island	+
:Island	:locOnIsland-	:City	t_City_locOnIsland	City	+
:Island	:locOnIsland-	:Mountain	t_Mountain	locOnIsland	-
:Island	:locOnIsland-	<unknown>	TOP_locOnIsland	GeoThing	+
:City	:locatedIn	:Country	t_City	locIn_Country	+
:City	:locatedIn	:Province	t_City	locIn_Province	+
:Province	:locatedIn	:Country	t_Province	locIn_Country	+
:City	:locatedIn		t_locatedIn	Area	+
:Mountain	:locatedIn		t_locatedIn	Area	+
:River	:locatedIn		t_locatedIn	Area	+
:Sea	:locatedIn		t_locatedIn	Area	+
:Province	:locatedIn-	:City	t_City	locIn_Province	-
:Country	:locatedIn-	:City	t_City	locIn_Country	-
:Country	:locatedIn-	:Province	t_Province	locIn_Country	-
:Province	:locatedIn-	<unknown>	t_locatedIn	GeoThing	+

:Country	:locatedIn-	<unknown>	t_locatedIn	GeoThing	+
:Continent	:locatedIn-	<unknown>	t_locatedIn	GeoThing	+
:Sea	:mergesWith	NO ENTRY!	t_mergesWith	Sea2	+
:Sea	:mergesWith	NO ENTRY!	t_mergesWith	Sea1	+
:Sea	:mergesWith-	NO ENTRY!	t_mergesWith	Sea2	+
:Sea	:mergesWith-	NO ENTRY!	t_mergesWith	Sea1	+
:Country	:encompassed	(:Continent)	t_EncompBy	encCont	-
:Continent	:encompassed-	(:Country)	t_EncompBy	countryEncBy-	-
:City	:inCountry	(:Country)	t_City	locIn_Country	+
:City	:inProvince	(:Province)	t_City	locIn_Province	+
:Province	:belongsTo	(:Country)	t_Province	locIn_Country	+
:Country	:inCountry-	:City	t_City	locIn_Country	-
:Country	:inCountry-	:Province	t_Province	locIn_Country	-
:Country	:belongsTo-	(:Province)	t_Province	locIn_Country	-

RangeClassJoinAttrs		
Table	**Column**	**RangeTable**
t_Country	capital	t_City
t_Province	capital	t_City
TOP_capital	id	t_Province
TOP_capital	id	t_Country
TOP_capital	capital	t_City
t_River	flowsInto_River	t_River
t_River	flowsInto_Lake	t_Lake
t_River	flowsInto_Sea	t_Sea
t_Lake	flowsInto_River	t_River
t_Lake	flowsInto_Lake	t_Lake
t_Lake	flowsInto_Sea	t_Sea
TOP_flowsInto	Water	t_River
TOP_flowsInto	Water	t_Lake
TOP_flowsInto	Water	t_Sea
TOP_flowsInto	flowsInto	t_River
TOP_flowsInto	flowsInto	t_Lake
Mountain	locOnIsland	t_Island
City_locOnIsland	City	t_City
City_locOnIsland	Island	t_Island
TOP_locOnIsland	GeoThing	t_City
TOP_locOnIsland	GeoThing	t_Mountain
TOP_locOnIsland	Island	t_Island
t_City	locIn_Country	t_Country
t_City	locIn_Province	t_Province
t_Province	locIn_Country	t_Country

t_locatedIn	GeoThing	t_City
t_locatedIn	GeoThing	t_Province
t_locatedIn	GeoThing	t_Country
t_locatedIn	GeoThing	t_Mountain
t_locatedIn	GeoThing	t_River
t_locatedIn	GeoThing	t_Sea
t_locatedIn	Area	t_Province
t_locatedIn	Area	t_Country
t_locatedIn	Area	t_Continent
t_locatedIn	Area	t_Desert
t_locatedIn	Area	t_Sea
t_mergesWith	Sea1	t_Sea
t_mergesWith	Sea2	t_Sea

NMJoinAttrs			
Class	**NMTab**	**LookupAttr**	**FKJoinAttr**
River	TOP_flowsInto	flowsInto	Water
Lake	TOP_flowsInto	flowsInto	Water
River	TOP_flowsInto	Water	flowsInto
Lake	TOP_flowsInto	Water	flowsInto
Sea	TOP_flowsInto	Water	flowsInto
:City	City_locatedIn_Province	Province	City
:Province	City_locatedIn_Province	City	Province
:City	City_locatedIn_Country	Country	City
:Country	City_locatedIn_Country	City	Country
:City	Mountain_locatedIn_Country	Country	Mountain
:Country	Mountain_locatedIn_Country	Mountain	Country
:City	TOP_locatedIn	Area	GeoThing
:Mountain	TOP_locatedIn	Area	GeoThing
:River	TOP_locatedIn	Area	GeoThing
:Province	TOP_locatedIn	GeoThing	Area
:Country	TOP_locatedIn	GeoThing	Area
:Continent	TOP_locatedIn	GeoThing	Area
<unknown>	TOP_locatedIn	GeoThing	Area
:City	City_locOnIsland	Island	City
:Island	City_locOnIsland	City	Island
<unknown>	TOP_locOnIsland	Island	GeoThing
:Island	TOP_locOnIsland	GeoThing	Island
:City	t_locatedIn	Area	GeoThing
:Province	t_locatedIn	Area	GeoThing
:Country	t_locatedIn	Area	GeoThing
:Mountain	t_locatedIn	Area	GeoThing

:River	t_locatedIn	Area	GeoThing
:Lake	t_locatedIn	Area	GeoThing
:Province	t_locatedIn	GeoThing	Area
:Country	t_locatedIn	GeoThing	Area
:Continent	t_locatedIn	GeoThing	Area
:Sea	t_locatedIn	GeoThing	Area
:Desert	t_locatedIn	GeoThing	Area
:Sea	t_mergesWith	Sea2	Sea1
:Sea	t_mergesWith	Sea1	Sea2
:Country	t_EncompBy	encCont	countryEncBy-
:Continent	t_EncompBy	countryEncBy-	encCont
:Water	t_locatedIn	Area	GeoThing

TopTables			
Property	**Table**	**LookupAttr**	**FKJoinAttr**
:name	TOP_name	name	oid
:population	TOP_population	population	oid
:area	TOP_area	area	oid
:capital	TOP_capital	capital	oid

8 Implementation and Processing: RDF/ER → Rel

To be flexible with experimenting with different variants, the following approach to the implementation is favoured:

1. implement the basic algorithm as described in Section 3.1,
2. implement the refinements discussed in Section 5 by operators on the schema and the metadata tables,
3. insert the tuples.

8.1 Lacking Functionality for TBox Analysis in SPARQL

SPARQL [4] 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. A set of such atomic built-in notions can be queried by SPARQL-DL [3].

In the following, even more complex queries are needed, that are even not possible in SPARQL-DL, 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].

The pure SPARQL semantics of the query is to match only if the above triples are explicitly contained in the ontology. When using a reasoner, triples are matched when they are implied in the model, but this only holds for individual triples – here, a certain combination is queried. Even more, usually triples with built-in OWL properties are even not stored explicitly, but translated into DL atoms for the reasoner. The same applies to SPARQL-DL, which only provides the possibility to query some of the latter built-in notions of OWL.
Thus, 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}.

8.2 Technicalities: SPARQL Expressiveness wrt. DL

There are several issues that prevent a straightforward realization of the algorithm:

- SPARQL cannot be used for querying on the DL level, e.g., whether something is functional or satisfies some cardinality constraint. One can (at most) query whether the respectives RDF triples are syntactically present in the model.
  Thus, for applying *reasoning* on that level, one must create appropriate classes and check for subclass relationships.
- The generation of RDF tuples is possible in SPARQL with the construct clause. But in SPARQL 1.0 it is not possible to generate new URIs by string manipulation and to turn them into resource URIs. With SPARQL 1.1 this will be possible, but pellet/jena do not yet support it.

Thus, the processing in such situations is as follows:

1. state a SPARQL query against the metadata (and derived intermediate results), "export" relevant combinations of, e.g., (domain, property, range) as RDF descriptions.
2. process each of the RDF descriptions by an operational language that creates auxiliary OWL class definitions that can be used to check whether the given combination satisfies some condition.
3. generate a model of the metadata and of the auxiliary definitions and state an appropriate SPARQL query that returns the results.

For Step (2), any language that allows to embed queries against RDF data and that supports string operations (to write OWL class definitions and to generate URIs) is suitable. In the prototype, a rather ugly scripting approach is taken, using a Makefile and several sed scripts (that work fully syntactically on RDF/N3

files using regular expression matching – any change in the order of serialization would break them). The sed scripts have finally to be replaced by a Java-based integrated program that uses SPARQL querying instead of pure regular expression matching.

8.3 SPARQL Chain Part I: Basic Algorithm

The file mondial-meta.n3 contains the basic ontology. Apart from the usual RDF terminology, the following meta notions are used as defined in mondial-er.n3:

er:Concrete the non-abstract classes.

er:Abstract all abstract and non-abstract classes.

er:Interface all abstract and non-abstract classes.

er:ReifiedRelationship all classes that represent reified relationships (subclass of er:Concrete).

er:reifies a property that describes that a :ReifiedThing reifies a rdf:Property.

:AddedInverseProperty additional inverse properties that have been assigned an explicit synthetic name.

```
#@prefix : <http://www.semwebtech.org/mondial/10/meta#>.
@prefix : <f://m#>.
@prefix er: <f://er#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.

# ER model notions
er:isa a owl:AnnotationProperty.
er:reifies a owl:AnnotationProperty.
er:onDomain a owl:AnnotationProperty.
er:onProperty a owl:AnnotationProperty.
er:onRange a owl:AnnotationProperty.
er:Entity a owl:Class; er:isa er:Abstract.
er:EntityType owl:disjointUnionOf (er:Class er:Interface).
er:Interface rdfs:subClassOf er:EntityType.
er:Class a owl:Class; rdfs:subClassOf er:EntityType;
   owl:disjointUnionOf (er:Concrete er:Abstract).
er:Concrete rdfs:subClassOf er:Class.
er:Abstract rdfs:subClassOf er:Class.
er:ReifiedRelationship rdfs:subClassOf er:Concrete.
er:SymmetricReifiedRelationship  rdfs:subClassOf er:ReifiedRelationship.
er:hasFunctProp a owl:AnnotationProperty.
er:hasCard2Prop a owl:AnnotationProperty.

# ... and now to the Mondial ontology

:MondialThing er:isa :Abstract; rdfs:subClassOf er:Entity.

## geometric aspects: Place (long,lat,elev.),
# Area (area, bordering), Line (length,between)
# note: small areas like city, lake, island, desert are also
# Places and Areas
:Geometrical rdfs:subClassOf er:Entity.
:Location  er:isa er:Interface.
:Area er:isa er:Interface.
:Place er:isa er:Interface.
:LargeArea er:isa er:Interface.
:Line er:isa er:Interface.

######## Concrete Classes

:Country er:isa er:Concrete.
:Province er:isa er:Concrete.
:City er:isa er:Concrete.
:Organization er:isa er:Concrete.
```



```
:Continent er:isa er:Concrete.
:Lake er:isa er:Concrete.
:Sea er:isa er:Concrete.
:River er:isa er:Concrete.
:Source er:isa er:Concrete.
:Estuary er:isa er:Concrete.
:Mountain er:isa er:Concrete.
:Desert er:isa er:Concrete.
:Island er:isa er:Concrete.
:Language er:isa er:Concrete.
:Religion er:isa er:Concrete.
:EthnicGroup er:isa er:Concrete.
```

Another file just declares some auxiliary annotation properties that are used *during* the extraction process:

```
@prefix aux: <f://a#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.

# aux notions
aux:isa a owl:AnnotationProperty.
aux:onDomain a owl:AnnotationProperty.
aux:onProperty a owl:AnnotationProperty.
aux:onRange a owl:AnnotationProperty.
```

[Filename: mondial-aux.n3]

The following subsections describe the SPARQL queries for extracting and reorganizing the schema information for applying the above mapping in order to create a relational database.

8.3.1 Auxiliary: Create Names for Unnamed Inverses With SPARQL, only named resources can be bound as answers. Thus, for handling properties in the SPARQL chain, all inverses have to be named resources. For this, for all inverses that are not yet named, a synthetic name is created.

```
# prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
 prefix : <f://m#>
 prefix er: <f://er#>
 prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
 prefix owl: <http://www.w3.org/2002/07/owl#>

# call  jena -q -pellet -qf missing-inverses.sparql -of mondial-missing-inverses.n3

 construct { [ er:isa er:MissingInverse;
               er:onProperty ?P ] }
# select ?P ?R

 from <file:mondial-meta.n3>
 from <file:mondial-er.n3>
 where
 {
    ?P a owl:ObjectProperty; rdfs:range ?R.
#    ?P a rdf:Property; rdfs:range ?R.  # use owl instead of rdfs.
#    ## do not consider xsd: datatypes
#    ?R a owl:Class .
#    FILTER (?P != owl:bottomObjectProperty && ?P != owl:bottomDataProperty)
    FILTER (?P != owl:bottomObjectProperty)

    OPTIONAL {?P rdfs:range ?R2 . ?R2 rdfs:subClassOf ?R . FILTER (?R2 != ?R)}
    FILTER (!bound(?R2))

    OPTIONAL {?P owl:inverseOf ?Q}
    FILTER (!bound(?Q))
}
order by ?P
```

[Filename: missing-inverses.sparql]

Creates auxiliary RDF descriptions that state for which object properties the inverse does not have an explicit name.

Example:

```
[] er:isa er:MissingInverse; er:onProperty :locatedIn .
```

Then create the missing N3 triples with synthetic names *prop*_Inv:

```
# call   sed -f createInvDefs.sed mondial-missing-inverses.n3 > mondial-invdefs.n3
# leave prefix lines unchanged
/@prefix/ p;
/@prefix/ d;
# leave empty lines unchanged
/^$/ p;
/^$/ d;
# add one more line to the already read one:
N;
# and match
s/^\[\]\s*er:onProperty :\([-_0-9a-Z]*\)\s*;\s*\n\s*er:isa\s*er:MissingInverse ./:\1_In
```

[Filename: createInvDefs.sed]

Example:

```
:locatedIn_Inv a owl:ObjectProperty; er:isa er:AddedInverseProperty;
  owl:inverseOf :locatedIn.
```

Whenever `mondial-meta.n3` is used in the following, the file `mondial-invdefs.n3` has also to be input.

8.3.2 General Signature of each Property For each property, its domain and range is listed: the most special :Abstract schema notion is used (without distinguishing :Abstract here, blank nodes for union classes would be used!).

```
# prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
 prefix : <f://m#>
 prefix er: <f://er#>
 prefix aux: <f://a#>
 prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
 prefix owl: <http://www.w3.org/2002/07/owl#>

# call  jena -q -pellet -qf createGeneralSignatures.sparql -of mondial-signatures.n3

# goal: General signature for each property using abstract or non-abstract
#   classes as domain/range (otherwise blank nodes are used!)

 construct { [ a aux:GeneralSignature;
               aux:onDomain ?C; aux:onProperty ?P; aux:onRange ?R] }
# select distinct ?C ?P ?R

 from <file:mondial-meta.n3>
 from <file:mondial-er.n3>
 from <file:mondial-invdefs.n3>
 where
 {
   ?C er:isa ?ERC . ?ERC rdfs:subClassOf er:Class.
   ?P a rdf:Property; rdfs:domain ?D ; rdfs:range ?R.
   ?D rdfs:subClassOf ?C.
   FILTER (?P != owl:bottomObjectProperty && ?P != owl:bottomDataProperty)

   ## the following is important if ?P rdfs:subProperty ?P2 is given and ?P
   ## has a stricter domain
   ##  (then ?P rdfs:domain ?D lists both domains (e.g.: mergesWith))
   OPTIONAL {?P rdfs:domain ?D2 . ?D2 rdfs:subClassOf ?D . FILTER (?D2 != ?D)}
   OPTIONAL {?P rdfs:range ?R2 . ?R2 rdfs:subClassOf ?R . FILTER (?R2 != ?R)}

   ## take the strictest ?C "above" ?D
   OPTIONAL {?C2 er:isa ?ERC2 . ?ERC2 rdfs:subClassOf er:Class.
             ?D rdfs:subClassOf ?C2 .
             ?C2 rdfs:subClassOf ?C . FILTER (?C2 != ?C)}

   FILTER (!bound(?C2))
   FILTER (!bound(?R2))
   FILTER (!bound(?D2))
}
order by ?P ?D
```

[Filename: createGeneralSignatures.sparql]

Example:

```
[] a aux:GeneralSignature ;
   aux:onProperty :locatedIn ; aux:onDomain :GeographicalThing ; aux:onRange :Area .
```

8.3.3 List of Tuples (Domain Class, Property, Range) For each pair (non-abstract-class, property) s.t. prop is defined for class, the range (as :Abstract) is named:

```
# prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
 prefix : <f://m#>
 prefix er: <f://er#>
 prefix aux: <f://a#>
 prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
 prefix owl: <http://www.w3.org/2002/07/owl#>

# call  jena -q -pellet -qf createClassProps.sparql -of mondial-class-props.n3

 construct { [ a aux:ClassPropRangeSignature;
               aux:onDomain ?C; aux:onProperty ?P; aux:onRange ?R] }
# select ?C ?P ?R ?D

 from <file:mondial-meta.n3>
 from <file:mondial-er.n3>
 from <file:mondial-invdefs.n3>
 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)

#   ?C rdfs:subClassOf ?D.
    ## eigentlich muesste man "intersection may be non-empty" testen
    # also: if (C disjoint D) not provable
    NOT EXISTS {?C owl:disjointWith ?D }

    ## speziellsten moeglichen Domain und Range nehmen
    ## ?D may be a blank node (it is not that important here)
    NOT EXISTS {?P rdfs:domain ?D2 . ?D2 rdfs:subClassOf ?D . FILTER (?D2 != ?D)}
    NOT EXISTS {?P rdfs:range ?R2 . ?R2 rdfs:subClassOf ?R . FILTER (?R2 != ?R)}
}
order by ?C ?P
```

[Filename: createClassProps.sparql]
Example:

```
[] a aux:ClassPropRangeSignature ;
   aux:onDomain :City ; aux:onProperty :locatedIn ; aux:onRange :Area .
```

```
[] a aux:ClassPropRangeSignature ;
   aux:onDomain :River ; aux:onProperty :locatedIn ; aux:onRange :Area .
```

8.3.4 Collect Functional Properties for each Class For each class c, and each property such that the intersection of c and the domain of p is not empty, create classes that allow to check whether p is functional on c, and whether p may be defined for c:

```
# prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
 prefix : <f://m#>
 prefix aux: <f://a#>
 prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
 prefix owl: <http://www.w3.org/2002/07/owl#>

# call jena -q -pellet -qf createFctTests.sparql -of mondial-fct-tests.n3

 construct {
  ## intersectionOf is necessary, since otherwise, the same Restriction
  ## is used in several definitions, and the reasoner will annotate it
  ## with the data of *all* tests that use it
  [ owl:intersectionOf(?C [a owl:Restriction;  owl:onProperty ?P;
                           owl:maxCardinality 1]);
    aux:isa aux:Card1Test; aux:onDomain ?C; aux:onProperty ?P; aux:onRange ?R ] .
  [ owl:intersectionOf(?C [a owl:Restriction;  owl:onProperty ?P;
                           owl:allValuesFrom [ owl:complementOf ?R]]);
    aux:isa aux:NonEmptyTest; aux:onDomain ?C; aux:onProperty ?P; aux:onRange ?R ]
  }
# select ?C ?P ?R

 from <file:mondial-meta.n3>
 from <file:mondial-er.n3>
 from <file:mondial-invdefs.n3>
 from <file:mondial-class-props.n3>
 where
 {
  [ a aux:ClassPropRangeSignature;
       aux:onDomain ?C;  aux:onProperty ?P ; aux:onRange ?R ] .
 }
 order by ?C ?R
```

[Filename: createFctTests.sparql]
Example:

```
[] aux:isa aux:Card1Test;
   aux:onDomain :Country; aux:onProperty :name; aux:onRange xsd:string;
   owl:intersectionOf (:Country [ a owl:Restriction ;
```

```
                owl:onProperty :name;  owl:maxCardinality 1  ]) .
[] aux:isa aux:NonEmptyTest ;
   aux:onDomain :Country; aux:onProperty :name; aux:onRange xsd:string ;
   owl:intersectionOf (:Country [ a owl:Restriction ;
                        owl:onProperty :name;
                        owl:allValuesFrom [ owl:complementOf xsd:string ]]) .
```

8.3.5 Functional Properties for each Class For each class, all functional properties are listed, using the above checks:

```
# prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
 prefix : <f://m#>
 prefix aux: <f://a#>
 prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
 prefix owl: <http://www.w3.org/2002/07/owl#>

# call jena -q -pellet -qf  checkFctProps.sparql -of mondial-functional-props.n3

construct { ?C aux:hasFunctProp ?P }

# select ?C ?P ?R ?RCC

 from <file:mondial-meta.n3>
 from <file:mondial-er.n3>
 from <file:mondial-aux.n3>
 from <file:mondial-invdefs.n3>
 from <file:mondial-fct-tests.n3>
 where
 {
   ?C1T aux:isa aux:Card1Test;
   aux:onDomain ?C; aux:onProperty ?P; aux:onRange ?R .

   ?RCC aux:isa aux:NonEmptyTest ;
   aux:onDomain ?C ; aux:onProperty ?P ; aux:onRange ?R .

   ?C rdfs:subClassOf ?C1T.

   NOT EXISTS { ?C rdfs:subClassOf ?RCC }
 }
```

[Filename: checkFctProps.sparql]

Example:

```
  :City aux:hasFunctProp :name, :population, :longitude, :latitude, :elevation.
```

8.3.6 Relational Schema Info According to Section 3.1 Creates a list of all schema information that is needed to create the basic relational schema and the metadata tables according to Section 3.1:

– for each non-abstract class all its functional properties,

– table name and both attribute names for $n : m$ tables (if both direction are named, one of them is ignored),

– property name and both attribute names where a TOP table is needed.

```
# prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
 prefix : <f://m#>
 prefix er: <f://er#>
 prefix aux: <f://a#>
 prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
 prefix owl: <http://www.w3.org/2002/07/owl#>

# call jena -q -pellet -qf makeRelSchema.sparql -of mondial-rel-schema.n3

 construct { ?C aux:hasFunctProp ?P .
             [ :topTableFor ?P2 ; :topOnDomain1 ?C2; :topOnDomain2 ?R2] .
             [ :nmName ?NMP ; :nmOnDomain1 ?C3; :nmOnDomain2 ?R3 ] .
             ?C aux:hasCard2Prop ?Card2P . }

# select ?C ?P ?Card2P ?C2 ?R2

 from <file:mondial-meta.n3>
 from <file:mondial-er.n3>
 from <file:mondial-invdefs.n3>
 from <file:mondial-signatures.n3>
 from <file:mondial-functional-props.n3>
 where
 {
  { ?C aux:hasFunctProp ?P }
  UNION
    ### top tables
  { ?P2 a owl:FunctionalProperty .
    [ a aux:GeneralSignature ;   ;   ## from file mondial-signatures.n3
      aux:onDomain ?C2 ;
      aux:onProperty ?P2 ;
      aux:onRange ?R2 ] .
    EXISTS { ?C21 aux:hasFunctProp ?P2 . ?C22  aux:hasFunctProp ?P2 . FILTER (?C21 != ?C

    ## domain is not a single non-abstract class
    NOT EXISTS { ?C2 er:isa er:Class }
  }
  UNION
    ### n:m Relationships
  { [ a  aux:GeneralSignature ;   ## from file mondial-signatures.n3
      aux:onDomain ?C3 ;
      aux:onProperty ?NMP ;
      aux:onRange ?R3 ] .

      ## that are not functional and whose inverse is not functional
      NOT EXISTS { ?NMP a owl:FunctionalProperty }
      NOT EXISTS { ?NMP owl:inverseOf ?NMPI . ?NMPI a owl:FunctionalProperty }
      ## omit the manually added inverses
      NOT EXISTS { ?NMP er:isa er:AddedInverseProperty }
      NOT EXISTS { ?NMP owl:inverseOf ?PINV . FILTER ( str(?PINV) < str(?NMP) ) }

      # note: n:m props that are defined in both directions in the original ontology ar
      #       still listed in both directions!  ignore one direction when creating the

      ?C owl:sameAs ?NMP .   ## only for table output: list NMP in the C/class/tablenam

      ## omit props that also exist as reifiedProp or whose inverse exists as reifiedPr
      NOT EXISTS { ?RP er:reifies ?NMP}
      NOT EXISTS { ?RP2 er:reifies [owl:inverseOf ?NMP]}

      ## omit 1:2 props that result from SymmetricReifiedProps
```

Example:

```
:City aux:hasFunctProp :name, :population, :longitude, :latitude, :elevation.
[] :topTableFor :capital; :topOnDomain1 :Area ; :topOnDomain2 :City .
[] :nmName :locatedIn ; :nmOnDomain1 :GeographicalThing ; :nmOnDomain2 :Area .
```

8.3.7 1:1 Properties (Step (A2)) 1:1-properties are listed in `mondial-rel-schema.n3` in both directions. Step (A2) of the algorithm removes one of them: if one direction is partial and the other direction is total, the partial one is removed.

Create classes that allow to check which (Domain,1:1-Property) combinations are partial/total:

```
# prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
 prefix : <f://m#>
 prefix aux: <f://a#>
 prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
 prefix owl: <http://www.w3.org/2002/07/owl#>

# call jena -q -pellet -qf createTotalityChecks.sparql -of mondial-total-checkdefs.n3

construct { [ owl:intersectionOf (?C [a owl:Restriction;
                                      owl:onProperty ?P; owl:minQualifiedCardinality 1])
              aux:isa aux:TotalityCheck;
              aux:onDomain ?C ;
              aux:onProperty ?P ] }

# select ?C ?D ?P ?PINV

 from <file:mondial-meta.n3>
 from <file:mondial-er.n3>
 from <file:mondial-aux.n3>
 from <file:mondial-invdefs.n3>
 from <file:mondial-functional-props.n3>
 where
 {
   ?C aux:hasFunctProp ?P .
   ?P owl:inverseOf ?PINV .
   ?D aux:hasFunctProp ?PINV
  }
```

[Filename: createTotalityChecks.sparql]
Example:

```
[] aux:isa aux:TotalityCheck ;
   aux:onDomain :Estuary; aux:onProperty :hasEstuary- ;
   owl:intersectionOf (:Estuary [ a owl:Restriction ;
                  owl:onProperty :hasEstuary- ;
```

```
owl:minQualifiedCardinality 1 ]) .
```

defines a class whose instances have at least one filler for the hasEstuary property. Since River turns out to be a subclass of this class, hasEstuary is total River.

Then do the reasoning+query:

```
# prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
 prefix : <f://m#>
 prefix aux: <f://a#>
 prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
 prefix owl: <http://www.w3.org/2002/07/owl#>

# call jena -q -pellet -qf doTotalityChecks.sparql -of mondial-11props.n3

construct {
  ?CTotalPPartialPINV aux:has11PropTotalInvPartial ?P .
  ?CTotalPPINV aux:has11PropTotalInvTotal ?P .
  ?CPartialPPINV aux:has11PropPartialInvPartial ?P .
  ?CPartialPTotalPINV aux:has11PropPartialInvTotal ?P }

# select ?CC ?P ?R

 from <file:mondial-meta.n3>
 from <file:mondial-er.n3>
 from <file:mondial-aux.n3>
 from <file:mondial-invdefs.n3>
 from <file:mondial-total-checkdefs.n3>
 from <file:mondial-functional-props.n3>
 where
 {
   ?C aux:hasFunctProp ?P .
   ?P owl:inverseOf ?PINV .
   ?D aux:hasFunctProp ?PINV .

   ?TC1 aux:isa aux:TotalityCheck ;
      aux:onDomain ?C; aux:onProperty ?P .
   ?TC2 aux:isa aux:TotalityCheck ;
      aux:onDomain ?D; aux:onProperty ?PINV .

   { ?CTotalPPartialPINV owl:sameAs ?C .
        ?C rdfs:subClassOf ?TC1 . NOT EXISTS { ?D rdfs:subClassOf ?TC2 }}
   UNION
   {?CTotalPPINV owl:sameAs ?C .
        { ?C rdfs:subClassOf ?TC1 . ?D rdfs:subClassOf ?TC2 }}
   UNION
   { ?CPartialPPINV owl:sameAs ?C .
        NOT EXISTS { ?C rdfs:subClassOf ?TC1 } . NOT EXISTS { ?D rdfs:subClassOf ?TC2 }
   UNION
   { ?CPartialPTotalPINV owl:sameAs ?C .
        NOT EXISTS { ?C rdfs:subClassOf ?TC1 } . ?D rdfs:subClassOf ?TC2 }
 }
```

[Filename: doTotalityChecks.sparql]

Example: the class River has a property hasEstuary which is partial (some rivers run dry in a desert), but its inverse is total on the class Estuary (every estuary belongs to some river).

```
:River aux:has11PropPartialInvTotal :hasEstuary.
:Estuary aux:has11PropTotalInvPartial :hasEstuary- .
```

Step (A2) then uses `mondial-11props.n3` to decide which directions are removed.

8.3.8 Reified Properties List all entity types that represent reified properties (to be able to create SQL views for the reified property):

```
# prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
 prefix : <f://m#>
 prefix er: <f://er#>
 prefix aux: <f://a#>
 prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
 prefix owl: <http://www.w3.org/2002/07/owl#>

# call  jena -q -pellet -qf md-for-reified.sparql -of mondial-md-reified.n3

## r: C -> D is reified by R:
## R: CR -> C, DR -> D

construct { [ a aux:ReifiedSignature;
                aux:forProperty ?P; aux:reifiedRelation ?R;
                aux:withDomain ?C; aux:domainProp ?CR;
                aux:withRange ?D; aux:rangeProp ?DR ] }

# select ?P ?R ?CR ?C ?DR ?D

from <file:mondial-meta.n3>
from <file:mondial-er.n3>
from <file:mondial-invdefs.n3>
where
{
  ?R er:reifies ?P .
  ?P rdfs:domain ?C; rdfs:range ?D .
  ?CR rdfs:domain ?R; rdfs:range ?RC .
  ?C rdfs:subClassOf ?RC . ?C er:isa ?ERC . ?ERC rdfs:subClassOf er:Concrete.
  ?DR rdfs:domain ?R; rdfs:range ?D .
   ## D is allowed to be abstract
  FILTER (?CR != <http://www.w3.org/2002/07/owl#bottomObjectProperty>)
  FILTER (?DR != <http://www.w3.org/2002/07/owl#bottomObjectProperty>)
}

## Eintrag fuer (C,CR-) -> (T_R,c,-) und
## Eintrag fuer (D,DR-) -> (T_R,d,-) im MD nachsehen und
## ((C,r)(t_R,d,+) und ((D,r-)(t_R,c,+) ins MD eintragen.
## (C,t_R,d,c) und (D,t_R,c,d) in NMJ-Table eintragen
```

[Filename: md-for-reified.sparql]

todo: clean naming in er:

Example (currently empty; the 1:1 props are total):

```
[] a aux:ReifiedSignature ;
     aux:reifiedRelation :Membership ;
     aux:forProperty :isMember ;
     aux:domainProp :ofMember ;
```

```
aux:rangeProp :inOrganization ;
aux:withDomain :Country ;
aux:withRange :Organization .
```

8.3.9 Generation of the MD and Tables The generation of the actual Mapping Dictionary and of the database tables uses the following files that have been generated by the above chain: which files to use

- `mondial-meta.n3`: OWL-DL ontology.
- `mondial-er.n3`: ER-model style annotations for symmetric relations (mergesWith: symmetric $n:m$, border: symmetric reified $n:m$)
- `mondial-aux.n3`: declarations of auxiliary things.
- `mondial-invdefs.n3`: named inverses.
- `mondial-rel-schema.n3`: main input ro schema generation, outcome of the above processing.
- `mondial-class-props.n3.n3`: ranges/datatypes.
- `mondial-11props.n3`: additional information about 1:1 properties.

The actual queries are as follows : 21.3.2013

- Entity Tables (incl. reified) Columns:

```
## schema-func.sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT DISTINCT ?C ?P ?R
WHERE {
        ?C :hasFunctProp ?P .
        ?P rdfs:range ?R
}
```

- $n:m$ object-valued (2nd column is URI) tables:

```
## schema-nm-object.sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT DISTINCT ?C ?P ?PINV ?DT ?D1 ?D2
WHERE {
  ?A :nmName ?P ;
     :nmOnDomain1 ?D1 ;
     :nmOnDomain2 ?D2 .
  [] a :ClassPropRangeSignature ;
     :onClass ?C ;
```

```
      :onProp ?P ;
      :withRange ?DT .
  ?P a owl:ObjectProperty
  OPTIONAL { ?P owl:inverseOf ?PINV }
}
```

– multivalued literal-valued properties (2nd column is a datatype):

```
##  schema-nm-literal.sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT DISTINCT ?C ?P ?DT
WHERE {
  ?A :nmName ?P ;
     :nmOnDomain1 ?D1 ;
     :nmOnDomain2 ?D2 .
  [] a :ClassPropRangeSignature ;
     :onClass ?C ;
     :onProp ?P ;
     :withRange ?DT .
  ?P a owl:DatatypeProperty
}
```

– entity tables: remove one direction for 1:1 properties

```
TO BE UPDATED; query mondial-11props.n3
```

– Top Tables: TableName, Domain, Range, Property/InverseName

```
## schema-top.sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT DISTINCT ?P ?D ?R ?PINV
WHERE {
        [] :topOnDomain1 ?D ;
       :topOnDomain2 ?R ;
       :topTableFor ?P .
    OPTIONAL { ?P owl:inverseOf ?PINV }
}
```

revise: handle with reified and/or symmetric properties

– Symmetric $n:m$ relationships (reified and plain)

```
## schema-fix-symmetric-props.sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT DISTINCT ?C ?D
WHERE {
        ?C a :SymmetricReifiedRelationship;
       :reifies ?P .
        ?P rdfs:domain ?D;
       rdfs:range ?R
}
```

8.4 SPARQL Chain Part II: Refinements

8.4.1 List of Triples (Domain Class, Property, Range Class) All triples (non-abstract-domain-class, prop, non-abstract-range-classes) s.t. prop is defined for domain-class and has values of range-class are listed:

```
# prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
 prefix : <f://m#>
 prefix er: <f://er#>
 prefix aux: <f://a#>
 prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
 prefix owl: <http://www.w3.org/2002/07/owl#>

# call jena -q -pellet -qf createPropSubranges.sparql -of mondial-class-prop-subranges.

 construct { [ a aux:SubRangeSignature;
               aux:onDomain ?C; aux:onProperty ?P; aux:onRange ?R] }
# select ?C ?P ?R ?D

 from <file:mondial-class-props.n3>
 from <file:mondial-meta.n3>
 from <file:mondial-er.n3>
 from <file:mondial-aux.n3>
 from <file:mondial-invdefs.n3>
 where
 {
  { ### non-abstract ranges:
    ## (do also not consider xsd: datatypes)
    [ a aux:ClassPropRangeSignature;
      aux:onDomain ?C;  aux:onProperty ?P ; aux:onRange ?R ] .
    ?R er:isa  ?ERC . ?ERC rdfs:subClassOf er:Concrete .
  }
  UNION
  { ### abstract ranges: all subclasses
    [ a aux:ClassPropRangeSignature;
      aux:onDomain ?C;  aux:onProperty ?P ; aux:onRange ?D ] .

    ## do not consider xsd: datatypes
    ?D a owl:Class .
    ?R rdfs:subClassOf ?D.
    ?R er:isa  ?ERC2 . ?ERC2 rdfs:subClassOf er:Concrete .
    NOT EXISTS { ?R owl:disjointWith ?D }
  }
}
order by ?P
```

[Filename: createPropSubranges.sparql]
Example:

```
   [] a aux:ClassPropSubRangeSignature ;
      aux:onDomain :City ; aux:onProperty :locatedIn ; aux:onRange :Country .
   [] a aux:ClassPropSubRangeSignature ;
```

```
   aux:onDomain :City ; aux:onProperty :locatedIn ; aux:onRange :Province .
[] a aux:ClassPropSubRangeSignature ;
   aux:onDomain :City ; aux:onProperty :locatedIn ; aux:onRange :Desert .
```

8.4.2 Nonempty (Domain Class, Property, Range Class) This does the same as discussed in Section 8.3.5, now for $n : m$ properties:

```
# prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
 prefix : <f://m#>
 prefix aux: <f://a#>
 prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
 prefix owl: <http://www.w3.org/2002/07/owl#>

# call jena -q -pellet -qf createFctSubrangeTests.sparql -of mondial-fct-subrange-tests

construct {
  ## intersectionOf is necessary, since otherwise, the same Restriction
  ## is used in several definitions, and the reasoner will annotate it
  ## with the data of *all* tests that use it
  [ owl:intersectionOf(?C [a owl:Restriction;  owl:onProperty ?P;
                           owl:allValuesFrom [ owl:complementOf ?R]]);
    aux:isa aux:NonEmptyTest; aux:onDomain ?C; aux:onProperty ?P; aux:onRange ?R ] .
  [ owl:intersectionOf(?C [a owl:Restriction;  owl:onProperty ?P;
                           owl:maxCardinality 1]);
    aux:isa aux:FctTest; aux:onDomain ?C; aux:onProperty ?P; aux:onRange ?R ]
 }

# select ?C ?P ?R ?RC

 from <file:mondial-meta.n3>
 from <file:mondial-er.n3>
 from <file:mondial-aux.n3>
 from <file:mondial-invdefs.n3>
 from <file:mondial-class-prop-subranges.n3>
 where
 { [ a aux:SubRangeSignature ;
     aux:onDomain ?C ; aux:onProperty ?P; aux:onRange ?R ] .
 }
```

[Filename: createFctSubrangeTests.sparql]
Example:

```
[] aux:isa aux:NonEmptyTest ;   aux:onDomain :City ;
   aux:onProperty :locatedIn ;  aux:onRange :Desert ;
   owl:intersectionOf (:City [ a owl:Restriction ;
                  owl:onProperty :locatedIn;
```

```
                owl:allValuesFrom [ owl:complementOf :Desert ]]) .
[] aux:isa aux:FctTest ;       aux:onDomain :City ;
   aux:onProperty :locatedIn ; aux:onRange :Desert ;
   owl:intersectionOf (:City [ a owl:Restriction ;
                owl:onProperty :locatedIn; owl:maxCardinality 1]) .
```

8.4.3 Functional Properties wrt (Domain Class, Range Class)

```
# prefix mon: <http://www.se
 prefix : <f://m#>
 prefix aux: <f://a#>
 prefix rdf: <http://www.w3.
 prefix rdfs: <http://www.w3
 prefix owl: <http://www.w3.

# call jena -q -pellet -qf c

construct { [ a aux:FctClass
                aux:onDomain

# select ?C ?P ?R ?RCC

 from <file:mondial-meta.n3>
 from <file:mondial-er.n3>
 from <file:mondial-aux.n3>
 from <file:mondial-invdefs.
 from <file:mondial-fct-subr
 where
 {
   ?C1T aux:isa aux:FctTest;
   aux:onDomain ?C; aux:onPr

   ?RCC aux:isa aux:NonEmpty
   aux:onDomain ?C ; aux:onP

   ?C rdfs:subClassOf ?C1T.

   NOT EXISTS { ?C rdfs:subC
 }
```

[Filename: checkFctSubranges.sparql]
Example:

```
[] a aux:FctClassPropSubRangeSignature ; aux:onDomain :City ;
     aux:onProperty :locatedIn ; aux:onRange :Country .
```

8.4.4 Makefile The following Makefile runs the above process:

```
#
# Harvester/Mondial
#

saxonXQ = java -cp ~dbis/XML-Tools/saxon/saxon9.jar net.sf.saxon.Query
jena = java -jar ~dbis/SemWeb-Tools/JENA-API/semweb.jar

this:
## 1) create inverses for all object-valued properties:
$(jena) -q -pellet -qf missing-inverses.sparql -of mondial-missing-inverses.n3
sed -f createInvDefs.sed mondial-missing-inverses.n3 > mondial-invdefs.n3
## 2) for each property (domain, range):
$(jena) -q -pellet -qf createGeneralSignatures.sparql -of mondial-signatures.n3
## 3) for each pair (non-abstract-class,prop) s.t. prop is defined for class: range
$(jena) -q -pellet -qf createClassProps.sparql -of mondial-class-props.n3
## 4) for each result from (3) create functionality and non-empty test and evaluate ther
## 3+4.1 can be incorporated into a single script
$(jena) -q -pellet -qf createFctTests.sparql -of mondial-fct-tests.n3
$(jena) -q -pellet -qf checkFctProps.sparql -of mondial-functional-props.n3
## 6) create relational schema info according to Sec.2 Whitepaper:
#     classtabs with fct props, nm-tabs, toptabs for fct.
$(jena) -q -pellet -qf makeRelSchema.sparql -of mondial-rel-schema.n3
## 7) 1:1 props: list those that are partial (for step (A2))
$(jena) -q -pellet -qf createTotalityChecks.sparql -of mondial-total-checkdefs.n3
$(jena) -q -pellet -qf doTotalityChecks.sparql -of mondial-11props.n3

## 8) Reified Props:
$(jena) -q -pellet -qf md-for-reified.sparql -of mondial-md-reified.n3

#######  Above: sufficient for Algo in Sec. 2  ########################################
## 8) for each pair (non-abstract-class,prop): all non-abstract range classes
$(jena) -q -pellet -qf createPropSubranges.sparql -of mondial-class-prop-subranges.n3
## 9) check nonemptyness of (dom,prop,rge) combinations:
$(jena) -q -pellet -qf createFctSubrangeTests.sparql -of mondial-fct-subrange-tests.n3
$(jena) -q -pellet -qf checkFctSubranges.sparql -of mondial-fct-subranges.n3
```

[Filename: Makefile]

8.5 Operators on Schema and Metadata Tables

Input: class tables and $n : m$ tables together with MD etc. metadata tables.

- split-domain(*nmtable*): partition *nmtable* by non-abstract domain classes (resulting in $n : m$ tables *c_nmtable*);
- split-domain(*nmtable*,*class*): separates the partition for the given domain *class*;
- split-range(*nmtable*): the same for the non-abstract range classes (resulting in $n : m$ tables *nmtable_c*);

- split-range(*nmtable*,*class*): analogously;
- split-all(*nmtable*): partition according to domain and range classes (resulting in $n : m$ tables *c_nmtable_d*);
- addToClassTables(*nmtable*): if *nmtable* is functional, then add it to the respective domain classes.
  If it is only functional on some of the domain classes, add it for those to the respective domain classes and leave the remaining entries in *nmtable*.
- union-nm(*property*): collect all $n : m$ tables about *property* to a single $n : m$ table *t_property*.
- removeProperty(*property*): Removes *property* from all tables, and removes the respective entries from the mapping dictionary.
- union-all(*property*): collect all $n : m$ tables and class table columns about *property* to a single $n : m$ table *t_property*.
- union(*nmtable1*,*nmtable2*): merge two $n : m$ tables into the first one.
- make-nmtable(*class*,*property*): turn column *property* of class table *t_class* into an $n : m$ table.
- is-functional(*nmtable*): returns true if *nmtable* is functional.
- is-domain-functional(*nmtable*,*class*): returns true if *nmtable* is functional on domain class *class*.
- is-range-functional(*nmtable*,*class*): returns true if *nmtable* is functional for range class *class*.
- same functions for is-inverse-functional.
- functional-for-domains(*nmtable*): returns all non-abstract classes in the domain of *nmtable* for which it is functional.
- functional-for-ranges(*nmtable*): analogously.
- rename(*table*,*name*): rename table.
- in all operators, the original tables or columns that are not needed any longer are deleted.
- in all operators, the metadata tables are updated accordingly.

The following metadata has to be associated with each $n : m$ table for implementing the above operators:

- name of the property,
- list of its non-abstract domain classes,
- list of its non-abstract range classes,
- name of the least common superclass of the domain,
- name of the least common superclass of the range,

8.6 Data Loading

The RDF triples are then inserted. There are (at least) three different strategies:

- insertion of complete tuples via SPARQL queries,
- incremental insertion of triples via insertions and updates,
- 2-pass insertion of triples via TOP tables and subsequent bulk processing to final tables.

In any case, in the class-definite case (i.e., for each object its class is explicitly known by an (s rdf:type c) statement), the TOP tables will be empty finally.

8.6.1 Insertion via SPARQL Queries For each table, a SPARQL query is generated that returns all complete tuples (will null values if a value is not given) that are then inserted into the database.

Example 23 *For the t_Country(id,name,code,capital,population,area) table, the SPARQL query is*

```
select ?X ?N ?C ?Cap ?P ?A
where { ?X a :Country; :name ?N; :code ?C.
        optional { ?X :capital ?CAP} .
        optional { ?X :population ?P} .
        optional { ?X :area ?A} }.
```

and then, for each tuple of answer bindings, roughly

```
INSERT INTO t_Country VALUES(?X,?N,?C,?Cap,?P,?A)
```

(including appropriate null value handling) is executed.
For the t_locatedIn(GeoThing,Area) table, the SPARQL query is

```
select ?X ?A
where {?X :locatedIn ?A}
```

and then, for each tuple of answer bindings, roughly

```
INSERT INTO t_locatedIn VALUES(?X,?A)
```

is executed (there are no optional or null values), and then, all tuples describing locatedIn between cities and countries/province are deleted since they are stored in the city class table (this is easier to generate than adding another clause that constrains the domain class to the SPARQL query).

The obvious disadvantage is that the whole RDF database must first be loaded into a queryable RDF repository. In the following, two approaches are described that do not require an RDF repository; the first one is fully incremental, the second one is 2-pass.

8.6.2 Incremental Insertion of Triples The triples of the RDF input are processed incrementally (note that this is possible both from N3 input and from an RDF/XML stream).

– create TOP tables for all properties.

During insertion,

– whenever a triple (s p o) is inserted, lookup in the OC dictionary whether the class of s is already known.
  • If yes, store the triple according to the MD:

    - if p is functional, and there is already an (incomplete) tuple for s in the respective class table, then UPDATE the tuple; otherwise insert it (can be done via SQL MERGE)
    - if p is non-functional, insert (s, o) into the appropriate $n : m$ table.
- If no, store the triple as $(s\ o)$ in the TOP_p table.
- whenever a triple $(s$ rdf:type $c)$ is inserted, add $(s\ c)$ to the OC dictionary.
- Moving tuples from the TOP tables to the main tables can be done by either
    - whenever a triple $(s$ rdf:type $c)$ is inserted, move all statements about s from the TOP tables to the main tables, or
    - after finishing data loading, for all $(s\ x)$ statements in any TOP_p table, lookup s in the OC dictionary, and store $(s\ p\ x)$ in the appropriate main table.

8.6.3 2-Pass Insertion of Triples The 2-pass insertion is just slightly different from the incremental one:

– create TOP tables for all properties.

During insertion,

– whenever a triple $(s\ p\ o)$ is inserted, store the triple as $(s\ o)$ in the TOP_p table.
– whenever a triple $(s$ rdf:type $c)$ is inserted, add $(s\ c)$ to the OC dictionary.
– After processing all triples, they are distributed over the final tables: for all $(s\ x)$ statements in any TOP_p table, lookup s in the OC dictionary, and store $(s\ p\ x)$ in the appropriate main table. Note that this can be done purely in (dynamic) SQL using INSERT and MERGE statements.

9 Discovering actual cardinalities from Pure RDFS

This section deals with the case that RDF input data is given with pure RDFS schema, without cardinalities. In this case, the keys of entities are the URIs.

The actual relational model is transparent to the user in all cases where the user states his queries in SPARQL, and these are then internally mapped to the relational model. Only if the user states SQL queries directly, he must know about the concrete schema. In a data-oriented "best-effort" way, the relational schema (and the cardinalities) are discovered as described below. If required, the user is finally informed about the actual schema.

Note that RDFS does not have a notion of "inverse". In case an RDFS schema with some OWL axioms about inverses, but again without cardinalities is given, inverses can also be handled in the loading process.

9.1 Three-Pass Algorithm

The creates and populates the relational storage for an RDFS schema and an RDF database by starting with $n : m$ tables, and moving functional properties to class tables afterwards:

Algorithm load3Pass(rdfs,rdf):

1. for every property p, create an $n:m$ table $t\_p(c,p)$ or $t\_p(c,d)$,
2. insert all tuples into the appropriate $n:m$ table (if inverses exist: choose one direction and invert the other tuples).
3. collect all pairs (c,p) or (c,p^-) such that property p or p^- (can easily be checked by an SQL GROUP BY query with COUNT) is functional on which classes c,
4. for each class c in this collection, create a class table $t\_c(id,p_1,\ldots,p_{n_c})$ with all its functional properties,
5. incorporate all tuples (s,o) from each t_p such that s is an instance of a class c where p or p^- is functional on c in a tuple about s in t_c and delete $t\_p(s,o)$.
6. delete empty $n:m$ tables,
7. create the metadata tables appropriately.

Aside: the storage after Step (2) coincides with the vertically partitioned storage (every property is a separate relation).

9.2 Incremental Algorithm

Another possibility is the following algorithm that starts with the assumption that all properties are functional and drops this constraint on-demand:

Algorithm loadIncremental(rdfs,rdf):

1. create TOP tables for all properties in *rdfs*.
2. For every class c, the primary table t_c contains a *id* column, and columns for *all* properties (i.e., assumption: all properties are functional). Again, all object-valued properties are stored in both directions as p and p^- (inverse).
3. for all input tuples (s p o) in *rdf*, execute addtriple(s,p,o),

Note that RDFS does not have a notion of "inverse". In case an RDFS schema with some OWL axioms about inverse, but again without cardinalities is given, a subsequent step is to be applied:

Algorithm handleInverse: (Sketch) For each pair of properties p and p^-:

1. if (w.l.o.g.) p^- does not occur in the database so far created, then
   - just add the MD entries for p^-.
   - optionally: if p is not functional, check whether p^- is functional. If so, add a p^- column to the appropriate class tables, replace all tuples for p in the data tables by appropriate entries for p^- and delete all columns/$n:m$ tables for p. Apply the appropriate modifications to the metadata tables.
2. if both p and $p-$ occur in the database (independent whether actually some data is stored redundantly or not),

- check whether one of them is functional (this can only be the case if for one of them, there is no $n:m$ table; still then, the inverse direction entries must be checked not to break functionality).
- if yes, store all entries in the functional direction, delete the others,
- otherwise, take one direction (arbitrarily, w.l.o.g., p) and store all entries currently expressed via p^- in direction p (by applying addtriple(s,p,o) for them.

Algorithm addtriple(s,p,o):

1. if there exists an entry (s, c) in the OC table, then execute spo2classtab(s,c,p,o).
2. otherwise, i.e., if s is not yet contained in the OC table:
   - if $p =$ rdf:type
     - add (s, o) to the OC table, and
     - execute distribute(s,c);
   - otherwise (i.e., $p \neq$ rdf:type):
     - add (s, o) to the TOP table for p,
     - if the class c of s can be derived from the RDFS metadata together with (s p o), execute spo2classtab(s,c,p,o) distribute(s,c);
3. if o is not yet contained in the OC table and the class c' of o can be derived from the RDFS metadata together with (s p o) execute distribute(o,c'),

Algorithm distribute(s,c): called when the class c of s just became known: distributes all entries about s from any TOP table into the "final" tables:

1. execute spo2classtab(s,c,p,o) for all triples (s p o) such that (s, o) is in the TOP table TOP_p of any property p (and remove these entries from the TOP tables).

Algorithm spo2classtab(s,c,p,o): Incorporate (s p o) into the class table for class c:

1. if there is not yet a tuple for s, create one with all values null except for setting the value of column p to o and return.
2. if there is already a tuple for s, and its value of column p is null, set column p to o and return (Step 3).
3. otherwise (this is the first tuple of this class for which p is not functional) execute makeNMscpo(s,c,p,o).

Algorithm makeNMscpo(s,c,p,o): Property p of class c just turned out to be non-functional due to triple (s p o):

1. if there is not already a table $t\_p(c, p)$ (i.e., table $t\_p(\ldots)$ does not already exist from another class that intersects with the domain of p):
   - create $t\_p(c, p)$ or $t\_p(c, d)$ [5],

[5] either simply use p as second column name, or derive the least common superclass d of the range from the RDFS metadata.

2. add (s, o) to $t\_p(c, p)$,
3. copy all pairs from $\pi[id, p](t\_c)$ to t_p, and delete column p from t_c.
4. update the Mapping Dictionary accordingly: remove the entry $((c, p), (t\_c, p))$ and add $((c, p), (t\_p, p))$.

Theorem 2 *The above algorithms create the strictest relational model that fits with the ER model for the actual cardinalities.*

Proof *By cases: entity type, relationship type.*

References

1. P.P. Chen. The entity-relationship model — towards a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
2. Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 4th Edition*. Addison-Wesley-Longman, 2004.
3. Evren Sirin and Bijan Parsia. Sparql-dl: Sparql query for owl-dl. In *OWL Experiences and Directions Workshop (OWLED)*, 2007.
4. SPARQL 1.1 Query Language. `http://www.w3.org/TR/sparql11-query/`, 2012.