Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Number ZAi-MSC-2016-33

Master's Thesis
submitted in partial fulfilment of the
requirements of the course "Applied Computer Science"

Extraction of Ontological Metadata and Generation of an OBDA Mapping from a Relational Schema

Lars Runge

Institute of Computer Science

Bachelor's and Master's Theses
of the Center for Computational Sciences
at the Georg-August-Universität Göttingen

30. November 2016

Georg-August-Universität Göttingen
Institute of Computer Science
Goldschmidtstraße 7
37077 Göttingen
Germany
☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
www.informatik.uni-goettingen.de

First Supervisor: Prof. Dr. Wolfgang May
Second Supervisor: Dr. Lena Wiese

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Göttingen, 30. November 2016

Abstract

Connecting and querying multiple relational databases can be quite the challenge especially when those databases have varying schemas and are located in different database management systems. One possible solution to this problem is the use of a higher level language such as an ontology that describes the data items in the relational schemas of interest. The ontology can then be used to gain universal Ontology-Based Data Access (OBDA) over all covered databases. This thesis covers the RELMODELBUILDER *tool, which analyses a relational database and automatically extracts ontological mapping metadata from it to provide the ontology-based data access. The tool utilises for that a mixture of direct mapping and more advanced mapping approaches to deal for instance with attributed relationships and to generate broad class hierarchies. The output is a model-intern representation of the mappings that serves as a common join-point for our other tools that expand the functionality. For example the generated mappings can be used by the* QUERYCONVERTER *tool to translate SPARQL-queries based on the extracted ontological information to SQL-queries for the used relational database. Furthermore multiple such created mappings from different relational schemas can be aligned by the* SCHEMAMATCHER *tool to grant OBDA to all relational databases from just one common definition of ontological vocabulary.*

Contents

List of Abbreviations

OBDA	Ontology-Based Data Access	1
RDBMS	Relational Database Management System	1
DBMS	Database Management System	3
RDF	Resource Description Framework	6
W3C	World Wide Web Consortium	5
ER	Entity-Relationship	9
RM	Relational Model	9
OWL	Web Ontology Language	9
SQL	Structured Query Language	14
PK	Primary Key	14
FK	Foreign Key	14
XML	Extensible Markup Language	18
URI	Unified Resource Identifier	18
SPARQL	SPARQL Protocol and RDF Query Language	19
RDFS	Resource Description Framework Schema	20
XSD	XML Schema Definition	21

Chapter 1

Introduction

Accessing and treating relational databases from the view of an ontology can bring many benefits for the user. This chapter introduces for instance the difficulties of querying multiple databases and querying in general from an uninformed user perspective with only the tools available from a Relational Database Management System (RDBMS). The examined problems can be tackled in various ways, but this thesis will focus on the creation and usage of relational-to-ontology mappings as one possible solution. Ontology-Based Data Access (OBDA) is not a new concept for solving such problems, thus related work and our contributions to this topic will be discussed in Sections 1.2 and 1.3. At the end of this chapter the tools which were used in the process and the real-world data sets that were used to construct the test cases will be highlighted.

1.1 Querying multiple databases

When considering real-life applications for analysing data the data comes often from different sources. Therefore data integration from different databases plays a huge part. When it comes to using SQL queries for this, the user has to know the schema of every database he wants to query. This is not a problem if all databases use the same schema, but most of the time even databases storing the same information have only similar schemas with slight differences. For example we can look at two automobile manufacturers A and B. Both companies probably maintain databases containing information regarding their employees and the cars they manufacture. If now company A acquires company B, we can imagine that the databases will be merged for easier overview thus creating the need to connect both. But those databases will most likely have discrepancies in their schemas even if they describe the same data items. The most common types of discrepancies that can occur are as followed:

1. The simplest case are differences in the naming of tables and columns that contain the same information. Figure 1.1 illustrates that difference. Company A and B maintain each one table “Employees” containing all the information regarding their employees like the name and salary. In this instance the tables follow the same structre, but the naming of the column which contains the second name of the employee is different.

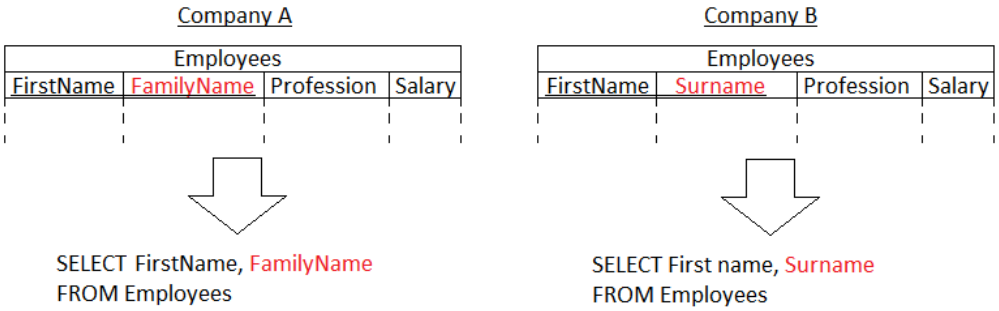


Figure 1.1: Influence of different column naming on the SQL query

2. The most important aspect are differences in the database schemas. For example this includes differences in the primary keys, which have to be represented in the query if the table has to be joined. Additionally information that is located in a single table in one schema can be split up into multiple tables in another schema. Both is represented in Figure 1.2. Company A retains its original "Employees" table with *First name* and *Family name* as its primary key. Both the employee's name and his salary can be found in the same table. On the other hand Company B has another schema that splits the employee details into two tables. One table "Employees" contains all the basic information about the employee like the name and address. An additional table named "EmployeeInformation" contains the job specific information of the employee e.g. the salary. Both tables are connected with a new primary key *EmployeeID* which additionally allows for multiple employees with the same name without overlap. When querying only for the employees names this would make no difference, but if the user also wants the salary Company B has to join both tables.

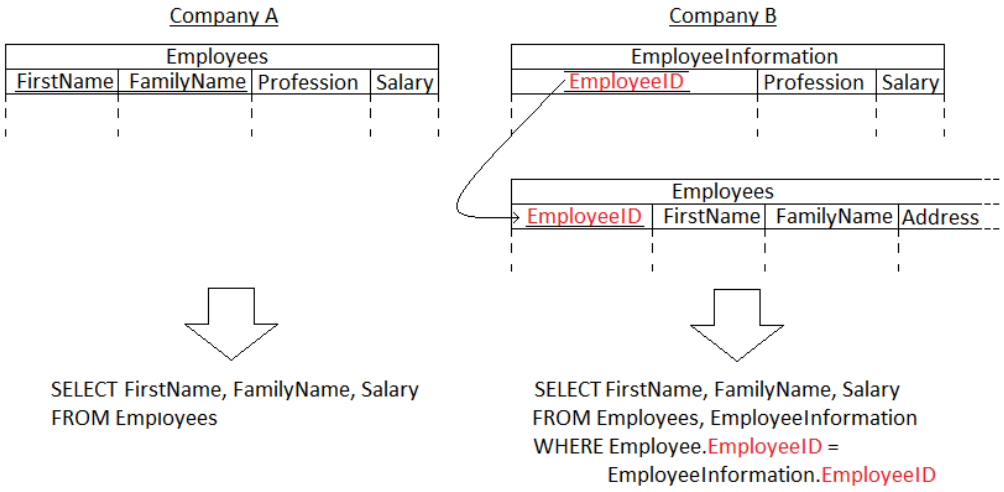


Figure 1.2: Influence of different schemas on the SQL query

3. Another factor that is possible are varying query syntaxes depending on the used Database Management System (DBMS). These emerge most commonly as different names for key words of the query. Figure 1.3 illustrates for instance that the key word *MINUS* in ORACLE systems is named *EXCEPT* in POSTGRES systems.

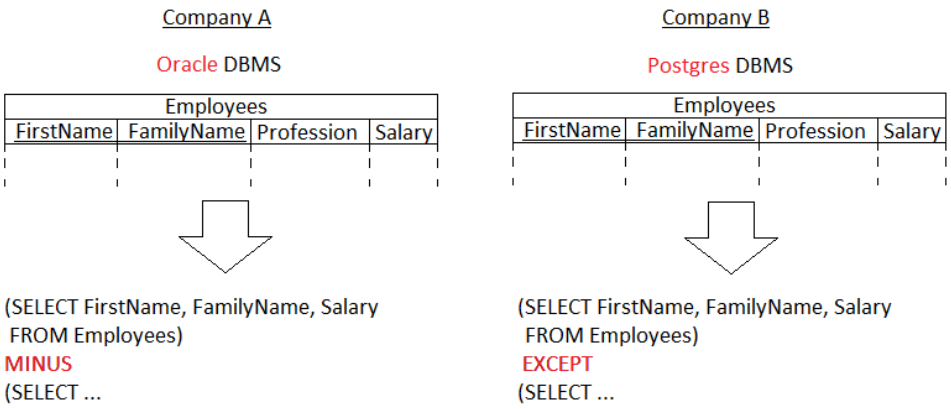


Figure 1.3: Influence of different DBMS syntax on the SQL query

A similar problem to the previous one but not as urgent occurs when a user wants to query a database which structure is not completely known to him. Maybe it is not necessary that the user has to learn the database schema for an one time use or even is not allowed to know the detailed structure due to security reasons.

The common solution to these problems is that the database administrator provides hand-written functions that give access to specified information based on the user requests. But depending on the number of occurences of such requests writing specified functions for each use-case can be quite the effort, especially with low cost-value ratios for one-time usage. On the other hand with our approach if a common higher-level access-point like an ontology is available to the user, it can be used to state queries to all databases at the same time which are then translated to specific SQL queries for each database. Taking the example from Case 2 again, Figure 1.4 sketches the process from the common ontology for the employee databases over a unified SPARQL query to the specific SQL queries for each database.

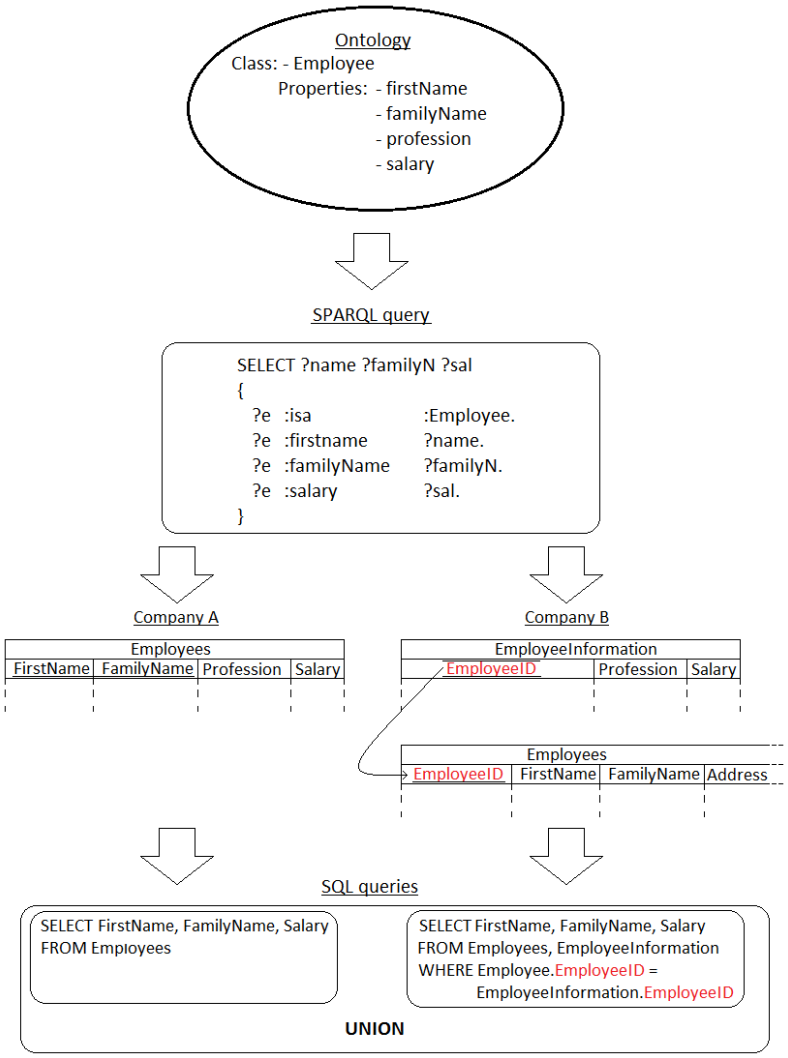


Figure 1.4: Concept of using a common ontology for unified access to all databases

1.2 Related Work

First of all to achieve OBDA to relational databases a suitable ontology for the database items and the respective mapping to the relational database has to be created. Doing so manually can be a time consuming task especially when a fully-fledged ontology with extensive class hierarchies should be created. However there also already exist some approaches that construct an appropriate ontology and the respective mappings in a automatic or semi-automatic way. The difference in these two approaches is that semi-automatic tools unlike fully automatic

ones require user input to guide the conversion process. Some prominent tools that create such mappings include BOOTOX [JRKZ+15], INCMAP [PBKH13], COMA++ [ADMR05], MIRROR [dMPC15], ONTOP [BCH+14] and KARMA [KSA+12].

BootOX is based on the World Wide Web Consortium (W3C) relational data to RDF direct mapping directives [W3Ca] to connect ontological vocabulary to the relational database items. Also it supports additional functionality like including an imported ontology to the bootstrapped one by aligning elements of both.

IncMap uses a semi-automatic approach that is able to map a given ontology to the relational database by using lexical and structural matching steps. It is however not able to construct a new ontology from the relational database that can be used for the mapping.

COMA++ is a schema and ontology matching tool, which offers a graphical interface to allow the user for instance to influence the matching process. It implements various different matching approaches like *Fragment-based Matching* and *Reuse-oriented Matching*.

MIRROR generates two groups of R2RML mappings from a relational database. The first group contains the usual mapping of a direct mapping approach, while the second one contains additional information M:N relationships and subclasses-of relationships.

ONTOP is a tool primarily for rewriting SQL queries to SPARQL ones available for instance as a Protégé plugin. However it also allows for limited ontology and mapping bootstrapping.

KARMA is a strict semi-automatic relational-to-ontology mapping generator that is specialised in integrating several data sources into one target ontology.

These six tools were recently tested by the RODI [PBJR+15] benchmarking tool. The tools were tasked to generate mappings for provided relational databases and target ontologies. Additionally for each such scenario a series of SPARQL queries had to be translated to the corresponding SQL format based on the generated mappings. The results were compared to the provided sample solutions. The tests were conducted on data sets from three different application domains such as conferences (CMT, SIGKDD, CONFERENCE), geodata (Mondial) and oil & gas exploration (NPD FactPages). The difficulty of the tests increased from the easier queries of the conference data sets asking only for atomic properties to the more difficult queries of the geodata and oil & gas domains that also select combinations of properties and additional constraints mimicking more realistic queries. Regardless of the approach of the tools, the general results of the benchmark were quite disappointing. While the test results for the atomic queries resulted in nearly acceptable outcomes, they concluded however that "all tested tools perform poorly on most of the more advanced challenges that come close to actual real-world problems". Especially for the MONDIAL and NPD FACTPAGES ontologies this is due to more advanced mapping challenges for instance "the introduction of a class hierarchies which groups data for several subclasses in a single table".

1.3 Contributions

We already created the mapping the other way around from ontology to relational tables in our previous work [RS13] using inter-model mapping metadata explained in Section 2.2.2. Therefore

we were curious if our approach could produce any better results as the previously mentioned programs. This task is split into two individual tools. On the one hand the RELMODELBUILDER program presented in this thesis which extracts an ontology from any relational database and creates suitable mappings. On the other hand the SCHEMAMATCHER [SCH16] which matches the created mappings from different relational databases to each other to achieve data integration. These tools utilise the same inter-model metadata from our previous work, which serves as the pivotal-point between them.

Thus our contributions for this task are the design and implementation of an algorithm to extract ontological metadata like classes and properties from the relational model. This extraction is explained in Section 3.2 and is based on the direct mapping approach of the W3C, but extended to a fine grained differentiation of relational table types. Our goal is not to produce a classic Resource Description Framework (RDF) ontology but the mapping metadata that is needed for our other tools. However such a classical ontology could be easily derived from the metadata entries if needed, because the mappings contain almost all the necessary information anyway. Furthermore we augment the OBDA mapping by elaborating a suitable class hierarchy as explained in Section 3.4 and expanding it as much as possible.

All in all we want to derive as much metadata information from the relational database as possible to produce a good foundation for our other tools like the QUERYCONVERTER or the SCHEMAMATCHER hightlighted in Section 2.2.3.

1.4 Tools

The implementation for this thesis was done in Java [Oraa] with JDK version 1.8.0_101. For the DBMS the standard POSTGRESQL [Groc] server in version 9.5 was used, which can be downloaded from [Grob]. The server automatically comes with the PGADMIN III tool, which provides a suitable interface to manage multiple databases as can be seen in Figure 1.5.

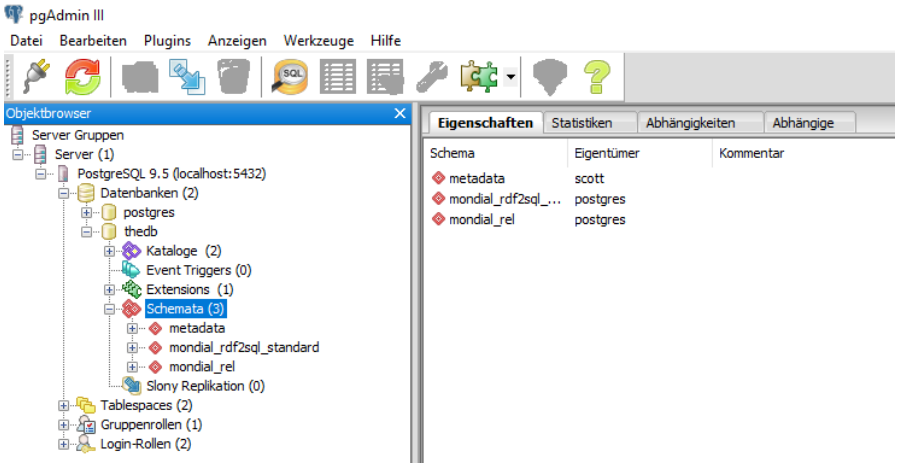


Figure 1.5: The PGADMIN III tool for managing POSTGRESQL databases

1.5 MONDIAL

The MONDIAL database is a project to accumulate geographical data from various sources and was initiated in 1998 as a use case of the F-Logic system FLORID. The data was mainly collected from the CIA World Factbook [Age], the 'Global Statistics', a predecessor of GEOHIVE [GEO] and a few other sources like the TERRA database from the University of Karlsruhe [May99]. The data set contains data about various geographical objects ranging from basic information about countries like their populations to international organizations and the location of their headquarters. The data is available for the data models F-Logic, SQL, XML, RDF/OWL and Datalog.

We use MONDIAL as the main test case for this thesis, because equivalent data sets for the data models SQL and RDF are necessary to verify the results. Both are accessible from [May]. Additionally slight variations of the schema are already available or can be constructed via other tools if needed. These different schemas cover a wide variety of connection types between the information providing even more test cases. We will cover the implementation and differences of the SQL and RDF version of MONDIAL in Section 2.1.

Chapter 2

Basics

This chapter covers the basic terminology and data structures that are used in this thesis. First of all the utilised data models will be discussed. In the beginning the conceptual Entity-Relationship (ER) model will be explained and how a Relational Model (RM) can be derived from it on the basis of the MONDIAL database example. After that the RDF data model will be discussed and how its structure can be utilised with Web Ontology Language (OWL) to formulate metadata information in an ontology. Finally the chapter will illustrate our *RelationalModel* structure, which contains a series of metadata tables describing the mappings from the RDF ontology to the relational database. Moreover it serves as a connection point for the output of the RELMODELBUILDER and the rest of our tools.

2.1 Data models

This section focuses on the data models that were used for constructing the MONDIAL data sets for relational or RDF databases. Most important for the to be generated mappings are the similarities and differences between the RDF ontology and the RM that will be explained based on the respective MONDIAL databases. These are also the foundation of all further examples.

2.1.1 Entity-relationship model

The ER [Che76] model is an conceptual data model that is able to graphically represent the relationship of things in a domain of interest. Among other things it is often used as visual representation for planning and describing a RM of a relational database.

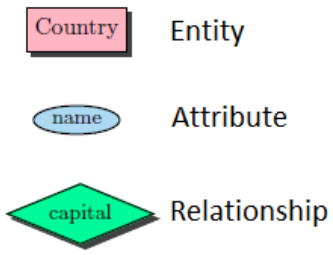


Figure 2.1: Basic building blocks of an ER model

The model mainly consists of *entity types*, *relationship types* and *attributes* that are visually represented as shown in Figure 2.1. Entities are the "things" of the world the user wants to describe, while attributes are the properties of an entity which they are described with. If two of these building blocks relate to each other, they are connected by a line. Figure 2.2 shows the *Country* entity with its basic attributes *name*, *code*, *population* and *area*. Each attribute can only have one value per instance of *Country* and the *code* attribute was defined as an *unique identifier* for this entity marked by an underscore. To easily distinguish the instances all values of *code* have to be unique to their respective instance.

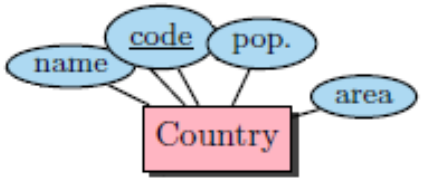


Figure 2.2: Country entity with attributes

Entities are connected to other entities not directly but by relationships. There are three different relationship types, which are distinguished by their cardinalities:

1. The first kind of relationship are *1:1* relationships like the *capital* relationship between the *Country* and *City* entities as shown in Figure 2.3. One country can only have one city as its capital and one city can only be the capital of one country.

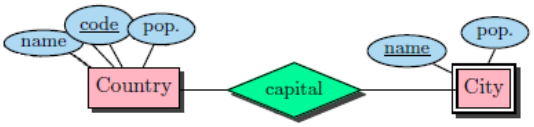


Figure 2.3: 1:1 relationship "capital" between Country and City

2. The second type of relationships is the *1:N* relationship. Figure 2.4 shows the *headq* relationship between the entities *City* and *Organization* as an example of such a relationship. An organization can only have one headquarter located in one city, while a city can host headquarters for multiple organizations.

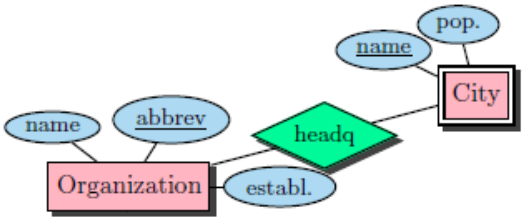


Figure 2.4: 1:N relationship "headq" for Organizations having their headquarter in Cities

3. The final type of relationships is the *N:M* relationship. An instance of this relationship in the MONDIAL domains is the *at* relationship between the entities *City* and *Lake* illustrated in Figure 2.5. One the one hand a city can be located at multiple lakes and on the other hand there can be multiple cities located at one specific lake.

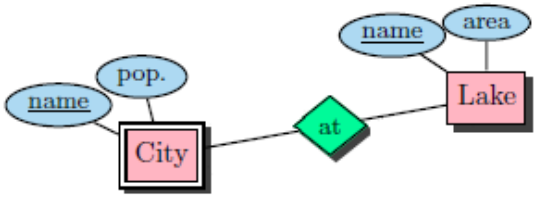


Figure 2.5: N:M relationship "at" describing Cities located at Lakes

Additionally it is important to note that attributes are not restricted to entities, but can be applied to all types of relationships too. Sometimes it is wise to further describe the relationship with attributes. For example with the *encompasses* relationship between *Country* and *Continent*. The *percent* attribute can be added to it as shown in Figure 2.6 that specifies how much of a country is encompassed by a specific continent.

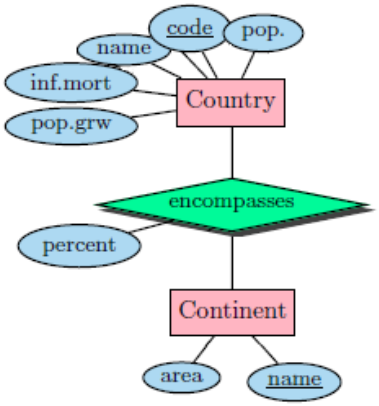


Figure 2.6: Relationship with attribute

Entities which can not be uniquely identified by its attributes alone and need to include the unique attributes from other entities are called *weak entities* and are marked with an additional frame as shown in Figure 2.7 with entity *Province*. A province name is only unique in the country it is located in, thus both the province name and the country code have to be combined to uniquely identify a province.

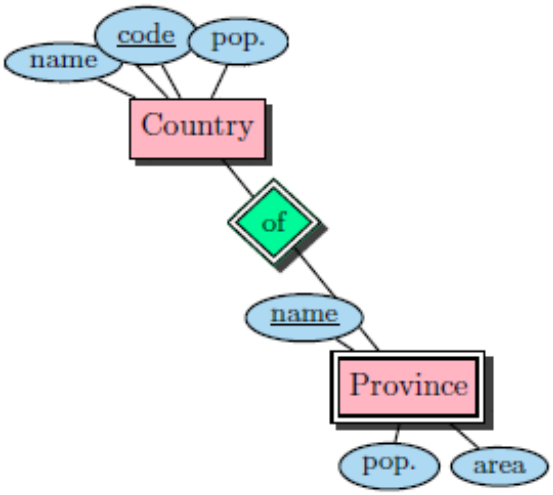


Figure 2.7: Weak entity Province

Following this approach the ER-diagram of the MONDIAL data set can be constructed from which the tables of the relational database are then derived later on. A slightly older but still representative diagram from 2015 is shown in Figure 2.8.

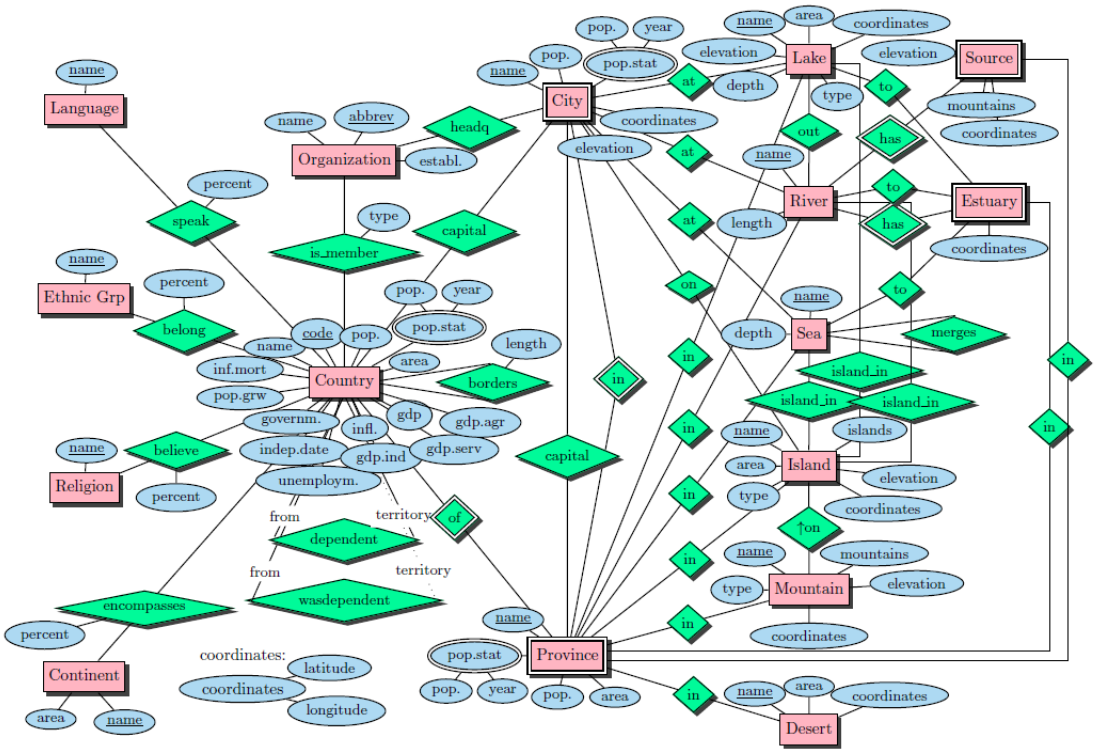


Figure 2.8: ER-diagram of the MONDIAL database

The ER is notably compound of:

- Entities: “Country”, “City”, “Province”, “Organization”, “Lake”, “Sea”, “River”, “Island”, “Mountain”, “Desert”, “Source”, “Estuary”, “Language”, “Ethnic Group”, “Religion”, “Continent”
- Relationships with attributes: “encompasses”, “borders”, “is_member”, “believe”, “belong”, “speak”
- N:M relationships:
  - 3x “at” from “City” to “Lake”, “Sea” and “River”
  - “on” from “City” to “Island”
  - 8x “in” from “Province” to “Lake”, “Sea”, “River”, “Island”, “Mountain”, “Desert”, “Source” and “Estuary”
  - “merges” from “Sea” to “Sea”
  - 3x “island_in” from “Island” to “Lake”, “Sea” and “River”

2.1.2 Relational database

RDBMS are the most common DBMS in todays business and research applications. They are called *relational* due to the fact that they are based on a so-called *relational model* defined by E. F. Codd [Cod70]. The relational model utilises *tuples* to represent data and *relations* to group such data tuples. For example a relation could be named "Country" and consists of the *attributes* "name","carcode", "population" and "capital". The attributes are paired with a data type to define which type of data an attribute can hold. In this case "name", "carcode" and "capital" would be of type *string*, while "population" is of type *integer*. Some tuples for the relation "Country" could for example stated like this:

$$Country = \{('Greece',' GR', 10816286,' Athina'), ('France',' F', 64933400,' Paris')\}$$

The relational model is a fully structured data model implying that all tuple that are grouped in the same relation have the exact same structure of attributes. Because of that, these relations can be represented as *tables* in a relational database. The table representing a relation like "Country" can adopt its attributes in form of *columns* and every tuple is entered as a *row* of the table. The table of the example relation and its tuples can be seen in Figure 2.9.

Country			
name	carcode	population	capital
Greece	GR	10816286	Athina
France	F	64933400	Paris

Figure 2.9: Simple table of relation "Country" and its tuples

In a RDBMS such a table can then be queried by using the Structured Query Language (SQL) [CB74]. The structure and use of SQL is not further covered in this section, because it does not lie in the focus of this thesis. Nevertheless a tutorial for the usage of SQL can be found on [Orab]. It is however important that a Primary Key (PK) is defined on the table to better reference a tuple in a relation for searching purposes and to avoid duplicate tuples in a relation. The columns belonging to the PK are defined to uniquely identify each tuple in the table and thus can not have a duplicate. For our example table this would be the "carcode" column. Additional to this functionality PKs open up the possibility to reference the tuples from other tables with so called Foreign Key (FK)s. Considering an additional relation "City" with the attributes "name" and "population" which stores all available cities with their respective populations. Among these cities are also the capitals of the countries. The PK of the "City" table is the column "name" and thus the "capital" column of the "Country" table can be considered a reference to those city entries. Such a reference is called a FK and is illustrated in Figure 2.10. To focus on the FK the depicted "City" table is only a simplified version of the actual MONDIAL relational database table for which the names of the cities are assumed to be globally unique.

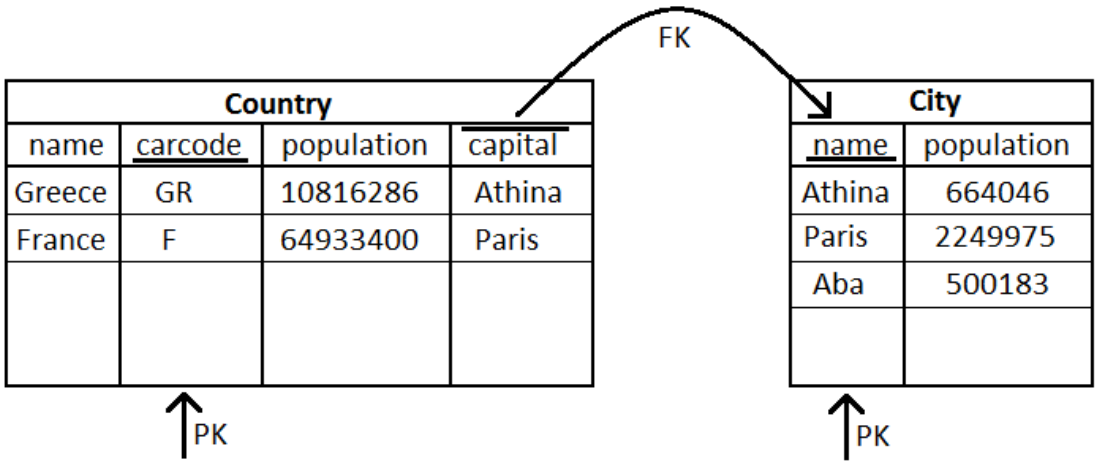


Figure 2.10: Tables “Country” and “City” with Primary Keys and Foreign Key

When constructing a relational database based on an already defined ER model, the entities, attributes and relationships can be converted to their relational database counterpart based on the following mappings. All mappings will be done with the examples defined in Section 2.1.1:

1. *Entities* are mapped to *tables* and their *attributes* to the *columns* of this table as shown in Figure 2.11 with the “Country” entity. The columns representing the uniquely indentifying attributes of the entity are defined as the PK of the table. If the entity is a weak entity, then the involved attributes of other entities for unique identification are added as FKs to the table. For instance the “City” entity which is a weak entity dependent on the “Province” entity, which in return is also dependent on the “Country” entity. In the end the PK of the “City” table has to be expanded to the columns “name”,”province” and “country”.

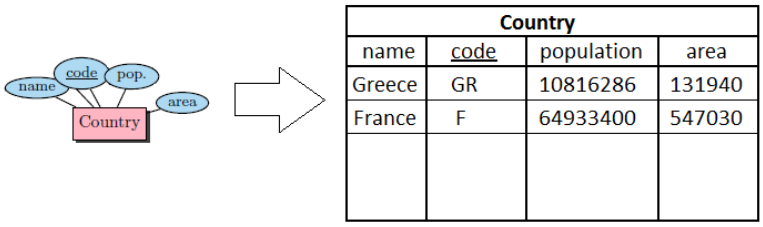
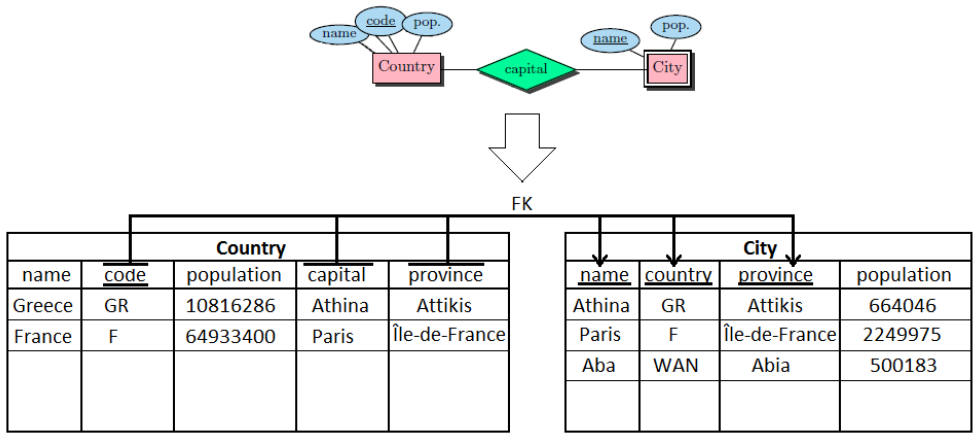


Figure 2.11: Mapping of Entities and attributes to the relational database

2. *1:1-relationship*s between two entities are represented as FKs from one side’s entity table to the other. To which side’s table the FK is added is left to the mapper and is often decided on the context of the relationship. Figure 2.12 shows the “capital” relationship again, which in this case is added to the “Country” table as a “capital” column. The other way around for instance it could have also be added to the “City” table as an “isCapitalOf” column. Every tuple from each table is only connected to one respective tuple of the other table.



Country

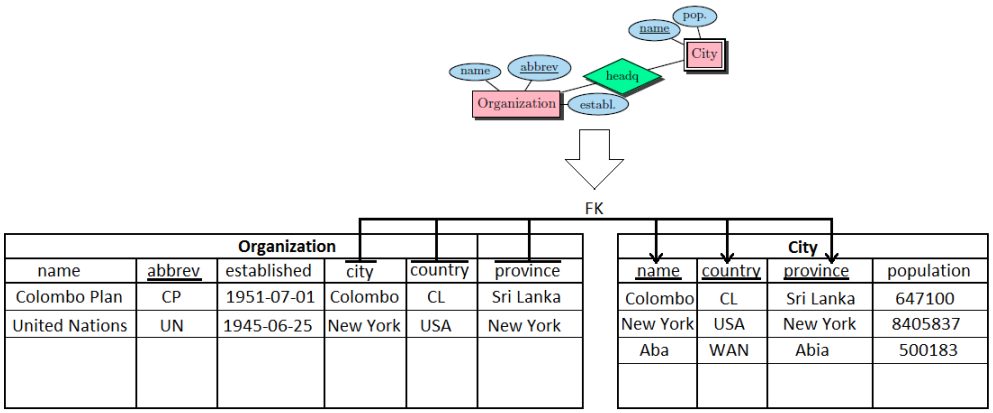
name	code	population	capital	province
Greece	GR	10816286	Athina	Attikis
France	F	64933400	Paris	Île-de-France

City

name	country	province	population
Athina	GR	Attikis	664046
Paris	F	Île-de-France	2249975
Aba	WAN	Abia	500183

Figure 2.12: Mapping of 1:1-relationships to the relational database

3. For *1:N-relationships* the mapping is quite similar to 1:1-relationships with the restriction that the FK has to be stored to the *functional* side (meaning the "1"-side) of the relationship. Thus in the example shown in Figure 2.13 the "headq"-relationship has to be stored in the "Organization" table. Each organization entry stores the unique location of its headquarters in a single column. If the relationship had been stored in the "City" table, then it would have had to be mapped to a series of columns each storing a potential organization that has its headquarters in an individual city. Due to the fact that there could be a huge amount of headquarters in a single city, this approach is not feasible.



Organization

name	abbrev	established	city	country	province
Colombo Plan	CP	1951-07-01	Colombo	CL	Sri Lanka
United Nations	UN	1945-06-25	New York	USA	New York

City

name	country	province	population
Colombo	CL	Sri Lanka	647100
New York	USA	New York	8405837
Aba	WAN	Abia	500183

Figure 2.13: Mapping of 1:N-relationships to the relational database

4. Following this argumentation *N:M-relationships* can not be stored in either table of the relationship's sides. Thus it has to be represented with an indivindual auxiliary table in a relational database. The auxiliary table only consists of columns representing the FKs to each side's entity table. This allows entries of either side to be related to multiple different entries of the other side. Figure 2.14 illustrates this mapping for the "at" relationship

between cities and lakes. However, the name “at” for a table would be quite improper and thus a name like “locatedAtLake” should be chosen.

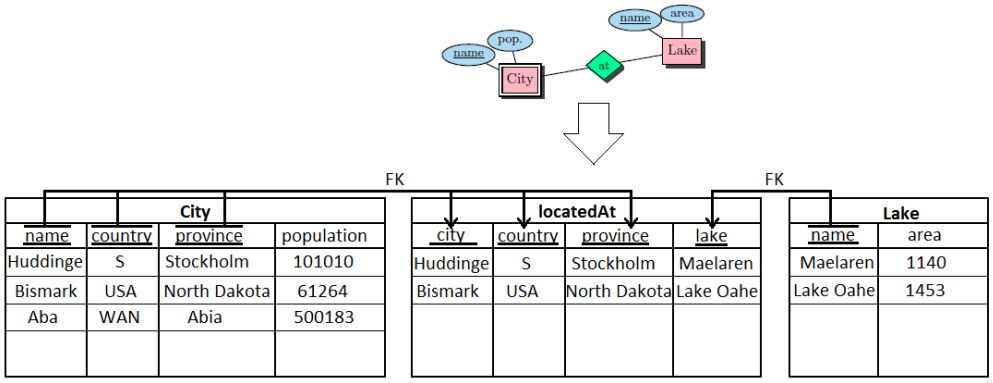


Figure 2.14: Mapping of N:M-relationships to the relational database

An exception of the previous mappings of relationships are attributed relationships. One the one hand, for N:M-relationships the additional attributes will just be added to the auxiliary table. On the other hand, for 1:1-relationships and 1:N-relationships the attributes could also be added to the functional side of the relationship where the FK will be stored. However to preserve the grouping of the relationship with its attributes it is recommended to separate the mapped columns to an individual auxiliary table similar to the mapping of N:M-relationships. When representing attributed relationships in relational databases with this approach, they are treated like entities in and of itself. This process is called *reification* with the attributed relationship being *reified* to an entity.

Overall the mentioned entities and relationships of the MONDIAL ER model shown in Figure 2.8 are represented as relational tables as mentioned above with the following exceptions:

1. Due to the lack of additional attributes beyond the PK of entities “Language”, “Ethnic Group” and “Religion”, they are not represented as individual tables with just one column. Instead the relationships with attributes “believe”, “belong” and “speak” are combined with their respective entity and are each represented as one table named after the original entity.
   For example the relationship “speak” becomes the table “Language” consisting of the columns “name”, “country” and “percentage”. The columns “name” and “country” are its PK with “country” being a FK to the “Country” table.

2. The eight tables representing the “in” relationships from “Province” are each named “geo_” and then the target entity. For example “geo_desert” and “geo_estuary”.

3. The three “at” relationships from “City” are combined in one table “located” with the columns “city”, “province”, ”country”, “river”, “sea” and “lake”.

4. The three “island_in” relationships from “Island” are combined in one table “islandin” with the columns “island”, “sea”, “lake” and “river”.

5. The “on” relationship from “City” was renamed into “locatedOn”.
6. The “on” relationship from “Mountain” was renamed into “mountainOnIsland”.

2.1.3 RDF database

The Resource Description Framework (RDF) [Groa] is a framework original intended for describing metadata information for web resources, but is now also used as a data model for databases. It has become a recommended standard by the W3C in 1999, which proceeds to update the standard whenever necessary. The most recent description from the 25th February 2014 of the RDF 1.1 specifications can be found on their website [W3Ce]. In Web sources the language Extensible Markup Language (XML) [W3Ci] is used to actually write data in RDF. To make it more human readable the *N3* notation [W3Cb] will be used throughout the thesis. This section focuses on the implementation of the RDF model to construct a database. Further explanation on how to use the data model to describe metadata information can be found in Section 2.1.4.

Unlike the relational model, the RDF model is not a fully structured data model, but a semi-structured one. This is because the modelling of data in RDF is always done through so-called *Triple*s. A triple is a tuple with three elements and consists of a *subject*, a *predicate* (also called *property*) and an *object* in this order. General speaking the predicate is like a connection from the subject to the object. This is equivalent to an *directed edge* in a *graph*, thus RDF is a *graph-based data model*. The subject is usually a “real world” object about which the user wants to store data, which is also called a *resource*. Lets take the country *Greece* as an example for such a subject. The predicate is similar to an attribute in the relational model and thus the *population* predicate for the subject *Greece* can be defined. Now as stated before the predicates point from the subject to the object in the triple, which therefore means that the *population* predicate points to the population number of *Greece.* In this case this would be the integer 10816286, which is the object of the triple. As is shown, unlike the subject, the object of a triple can be an object or a literal value. All in all the following triples can be stated:

$$(Greece, population, 10816286)$$

Due to the fact that the RDF model was originally designed to describe web resources it is recommended to give things that are described as subjects of a triple a Unified Resource Identifier (URI). These URIs are unique names for easier differentiation in the web and thus are often in the style of an URL. It was imagined that the URLs in most of the URIs would lead to actual web pages describing the the real world object or person the URIs stands for, but that is mostly not the case. Keeping this style, the resource *Greece* of our MONDIAL database would get an URI like:

$$< http://www.semwebtech.org/mondial/10/countries/GR/ >$$

The international vehicle registration code “GR” in the end of the URI is taken instead of the full name “Greece” to uniquely define the URI. Furthermore predicates are also described through an URI to differentiate them from equally named predicates from other sources. As such the *population* predicate can be named:

$$< http://semwebtech.org/mondial/10/meta\#population >$$

To avoid the tedious work of writing out the whole URL part of the URI over and over again it is possible to define prefixes for consistently used URLs in the beginning of the document. This functionality is illustrated in Figure 2.15, which also contains the triple of the remaining basic information about Greece from previous examples.

```
@prefix : <http://www.semwebtech.org/mondial/10/meta#>.

<http://www.semwebtech.org/mondial/10/countries/GR>   :name        'Greece'.
<http://www.semwebtech.org/mondial/10/countries/GR>   :carCode     'GR'.
<http://www.semwebtech.org/mondial/10/countries/GR>   :population  10816286.
<http://www.semwebtech.org/mondial/10/countries/GR>   :capital     <http://www.semwebtech.org/mondial/10/countries/GR/cities/Athina>.
```

Figure 2.15: Basic description of the Greece object in the RDF model

As the figure shows the prefix $< http://semwebtech.org/mondial/10/meta\# >$ is condensed in the simple character ":". When using the prefix for example with ":population" the name of the predicate is automatically inserted behind the "#" in the prefix. A triple statement usually ends with ".", but due to the fact that URIs for subjects and objects can not be condensed with prefixes it is recommended to use the character ";" instead to automatically reuse the previous subject. Thus we can write:

$$\begin{array}{lll} < http://www.semwebtech.org/mondial/10/countries/GR/ > & :name & 'Greece'; \\ & :carCode & 'GR'. \end{array}$$

The URIs of our user space will remain constant in the rest of the thesis, but will be shortened due to their length. A database in the RDF model then consists of a whole lot of those triples, but there are different strategies to store and manage them. The first option is to store all triples in one or multiple XML files, but this method becomes more inefficient the greater the amount of triples. Therefore there exist especially developed DBMS called triplestores that natively store and manage even huge amounts of triples. Similar to SQL for relational databases there also exists a query language for triples called SPARQL Protocol and RDF Query Language (SPARQL). SPARQL became the W3C recommendation for querying RDF data in January 2008 [W3Ch]. It is a highly flexible language that uses the triple format with user defined variables for subjects and objects to bind data items to. As before with SQL this section will not go into further details about SPARQL.

However if the user wants to use the highly optimized and standardized relational database to store these RDF triples, they have to be transformed to a suitable format. There are many ways to tackle this problem and the most naive approach would be to construct a single table with four columns as shown in Figure 2.16.

Triples			
subject	predicate	object	datatype
<.../countries/GR>	<.../meta#name>	Greece	String
<.../countries/GR>	<.../meta#population>	10816286	Number
<.../countries/GR>	<.../meta#capital>	<.../cities/Athena>	Object

Figure 2.16: Naive storing of triples in a relational database

The first two columns of the table are of type *VarChar* and store the subject and predicate respectively. The third column stores the object of the triple but due to the variance of datatypes that are used for objects the generic *blob* datatype has to be chosen in a relational table. Alternatively all values from the object could be converted to the *string* datatype in stored in a *VarChar* column. Regardless this is the reason a forth column has to be added to store the original datatype of the object from each triple. However this setup comes with a whole lot of problems. First of all is it not possible to search for entries in columns with datatype *blob*, which makes searching entries with specific object data difficult. But even with similar approaches that do not utilise the *blob* datatype searching and comparing data items is inefficient, if for instance characteristics of the original datatype can not be utilised or the table contains a huge amount of entries. Overall relational databases are just not optimized for the usage of a single big table. To better use the possibilites of the relational storage it is recommended to try to build a relational schema based on the metadata about the triples and then add the information to the tables. To express the metadata a RDF ontology is quite useful as explained in Section 2.1.4. Based on such an ontology the respective tables in a relational database can be automatically constructed using the RDF2SQL [RS14] program from our previous work. The basic approach of RDF2SQL and the produced tables will be explained in Section 2.2.1.

2.1.4 RDF ontology

An ontology for databases is general speaking a set of clauses that are used to describe which types of data items can exist in the specified environment. Ontologies can be used in every data model and thus are not specific to RDF. The ER diagram explained in the previous section can also be considered an ontology, but for the remaining thesis the focus lies on RDF when talking about ontologies.

An ontology for RDF can be constructed with the use of the Resource Description Framework Schema (RDFS) [W3Cf] and the OWL [W3Cc]. RDFS became the W3C recommendation for describing RDF vocabulary in February 2004 [W3Cg]. It provides a set of terms using the RDF model to describe basic elements of a RDF ontology. Most notably this includes the definition of *Classes* and *Properties* that will be explained later on. Unfortunately the modelling possibilities with only RDFS are quite limited and thus a new recommendation was developed by the W3C OWL WORKING GROUP named OWL that widely expands the terms from RDFS. The first version of OWL was also published in 2004 [W3Cd], but is since then considered closed. The newer version OWL 2 was published in 2009 and lastly updated in 2012. Because of the

huge extent of OWL 2 this section will only cover the functionality that is necessary for this thesis.

There are multiple ways to express an ontology, but the N3 notation will be continued to be used. This also opens up the use of XML Schema Definition (XSD), which is a recommendation of the W3C for the description of XML elements, to utilise its definition of basic datatypes. To utilise the pre-defined terms of RDFS, OWL and XSD it is beneficial to first define the prefixes of their respective language. These prefixes are shown in Figure 2.17 and are used for the rest of the thesis. The first three prefixes are to better differentiate the objects in the user space.

```
@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#>.
@prefix   xsd:   <http://www.w3.org/2001/XMLSchema#>.
```

Figure 2.17: Prefixes of the user space, RDF, RDFS, OWL and XSD

As said before one of the most important aspect of the ontology is the definition of *classes* that give a general structure to the data implementing the ontology. A class is defined via the object term *owl:Class* and can be assigned to a subject with the predicate *rdf:type* which is commonly abbreviated with the character *a*. For example a *Country* class can be defined and then the URI of the Greece subject assigned to be an *instance* of this class.

$$\begin{array}{rcl} :Country & a & owl:Class. \\ <.../country/GR/> & a & :Country. \end{array}$$

Defining various classes is not only helpful to differentiate different types of data items, but also gives context to the human reader. Furthermore *properties* of a class can be defined that describe which predicates an instance of the class must have. OWL gives us access to a wide variaty of property types. The most commonly used types are:

- *owl:DatatypeProperty*: Defining that the datatype of the value used with the predicate has to be a generic datatype.
- *owl:ObjectProperty:* Defining that the value used with the predicate is an instance of a class.
- *owl:FunctionalProperty*: Defining that the predicate is functional, i.e. there can be atmost one value for each instance.

A property can only be either a *DatatypeProperty* or an *ObjectProperty*, while the *FunctionalProperty* feature can be added optionally. Thus a *name* property can be defined as such:

$$\begin{array}{rcl} :name & a & owl:DatatypeProperty; \\ & a & owl:FunctionalProperty; \\ & rdfs:domain & :Country \\ & rdfs:range & xsd:string \end{array}$$

The domain describes of which class the subject has to be to use the property and the range describes what values the property can hold if used. We mainly differentiate properties in this thesis on which type of range they are defined. On the one hand, a *DatatypeProperty* ranges over a literal datatype e.g. string, number, date, etc. On the other hand, a *ObjectProperty* that ranges over a custom class is also called *object-valued*. Such a property of our *Country* class could be the *capital* property that links a *Country* resource to the appropiate resource of class *City* that is the capital of the country.

$$\begin{array}{lcl} :capital & a & owl:ObjectProperty; \\ & a & owl:FunctionalProperty; \\ & rdfs:domain & :Country \\ & rdfs:range & :City \end{array}$$

Data triples of both properties could look like this:

$$\begin{array}{lcl} :Country & rdf:type & owl:Class. \\ :City & rdf:type & owl:Class. \end{array}$$

$$\begin{array}{lcl} <.../country/GR/> & a & :Country. \\ <.../city/Athina/> & a & :City. \end{array}$$

$$\begin{array}{lcl} <.../country/GR/> & :name & "Greece"; \\ & :capital & <.../city/Athina/>. \end{array}$$

Properties are directed links from the subject to the object. This means that for properties of type *ObjectProperty* the reverse direction is also possible. The new property describing the reverse direction is called the *inverse* of the original property. Most inverses in the MONDIAL ontology are named like the original property with "_Inv" attached to the end, but the inverse of the *capital* property could also be called *isCapitalOf*. Inverses are defined by the *owl:inverseOf* property:

$$\begin{array}{lcl} :isCapitalOf & a & owl:ObjectProperty \\ :capital & owl:inverseOf & :isCapitalOf \end{array}$$

For inverses the domain and range are switched, which also includes that *DatatypeProperties* can not have an inverse, because the domain is not allowed to be a literal datatype. Another important aspect of an ontology is the definition of a class hierarchy via subclasses. Subclasses help to categorise classes in shared superclasses. But these superclasses are often not used explicitly in MONDIAL and are thus called *abstract classes*. On the other hand all classes that do have instances are called *concrete classes*. This can be simulated by defining the two classes *Abstract* and *Concrete:*

$$\begin{array}{lll} er:Abstract & rdf:type & owl:Class. \\ er:Concrete & rdf:type & owl:Class. \end{array}$$

And then use the standarized *rdfs:subClassOf* property to define sub- and superclasses. A good example for this are the three classes *Lake*, *Sea* and *River* that can be combined into the abstract superclass *Water*:

$$
\begin{array}{ccc}
:Water & a & er:Abstract. \\
:Sea & a & er:Concrete. \\
:Lake & a & er:Concrete. \\
:River & a & er:Concrete.
\end{array}
$$

$$
\begin{array}{ccc}
:Sea & rdfs:subClassOf & :Water. \\
:Lake & rdfs:subClassOf & :Water. \\
:River & rdfs:subClassOf & :Water.
\end{array}
$$

The whole MONDIAL ontology is then constructed with the use of this terminology, but is far too extensive to illustrate everything in this thesis.

2.1.4.1 Reified properties

An interesting special case of non-functional properties are *reified properties*, which are additionally linked with further properties similar to attributed relationships in the ER model. To better illustrate this concept the property *encompassed* from the MONDIAL RDF/OWL ontology is taken as an example. The definition of the *encompassed* and its inverse property *encompasses* is shown in Figure 2.18. The domain of the property is the abstract superclass *EncompassedArea*, whose definition is also highlighted in the Figure and is a combination of the concrete classes *Country* and *Province*. The range of *encompassed* is the concrete class *Continent*, which makes it an *owl:ObjectProperty*.

```
:encompassed     a                owl:ObjectProperty;
                 rdfs:domain      :EncompassedArea;
                 rdfs:range       :Continent;
                 owl:inverseOf    :encompasses.

:encompasses     a                owl:ObjectProperty.


:EncompassedArea      a                    owl:Class;
                      owl:equivalentClass [ owl:unionOf (:Country :Province) ].
```

Figure 2.18: Definition of properties *encompasses* and *encompassed*

All in all the function of the property is to relate all *Country* and *Province* resources to the *Continent* resources that they are encompassed by. The other way round, the *inverse* of the *encompassed* property named *encompasses* describes for each continent resource which countries and provinces it encompasses and is also non-functional. The property could then be used for the country Russia in the MONDIAL RDF database as follows:

$$< .../country/R/ > \quad : encompassed \quad < .../continent/Europe > .$$
$$< .../country/R/ > \quad : encompassed \quad < .../continent/Asia > .$$

But what if we additionally want to describe how many *percent* of a country are located in each continent? A single property can only relate two values together, thus to link together and easily access both the *encompassed* property and the *percent* property, the *encompassed* property has to be reified. The RDF model allows the definition of such reification, however there is currently no official standard to support this feature in an OWL ontology. To circumvent this restriction, we manually defined a *ReifiedRelationship* class and a *reifies* property to define such relationships for our internal processes.

```
:Encompassed    er:isa          er:ReifiedRelationship;
                er:reifies      :encompassed.


:encompassedArea      a               owl:ObjectProperty;
                      a               owl:FunctionalProperty;
                      rdfs:domain     :Encompassed;
                      rdfs:range      :EncompassedArea;
                      owl:inverseOf   :encompassedByInfo.


:encompassedBy  a               owl:ObjectProperty;
                a               owl:FunctionalProperty;
                rdfs:domain     :Encompassed;
                rdfs:range      :Continent.
                owl:inverseOf   :encompassesInfo;

:percent   a              owl:DatatypeProperty;
           a              owl:FunctionalProperty;
           rdfs:domain    :Encompassed;
           rdfs:range     xsd:decimal.
```

Figure 2.19: Definition of reified property *Encompassed* and its properties

Figure 2.19 illustrates the usage of these terms in the MONDIAL ontology to define the reified *Encompassed* class and its properties. *Encompassed* is defined to be an instance of the new *ReifiedRelationship* class so that it can be treated as a class of its own for the rest of the OWL definitions. The *reifies* property is there to store which original property is reified by the this class. Then two new properties for *Encompassed* are defined that each point towards one of both sides of the original relation. On the one side the property *encompassedArea* ranges over the *EncompassedArea* class, which was previously the domain-side of the *encompassed* property. On the other side the property *encompassedBy* has the *Continent* class as its range, which was the range-side of the *encompassed* property. Both new properties are functional properties, because each combination of $(EncompassedArea, Continent)$ is now treated as a resource of the *Encompassed* class with a unique URI. Finally the *percent* functional property can also be linked to it, which is a literal-valued property with range *xsd:decimal*.

A data item then uses the *Encompassed* class to express the connection between continent and country as shown in Figure 2.20. The *Encompassed* class instances in the MONDIAL RDF database

are just blank nodes, meaning they have no explicit URIs and only an internal unique identifier. This is done because the URIs of *Encompassed* resources should not be queried directly but rather be used for the new properties *encompassedArea*, *encompassedBy* and *percent*.

```
 [ rdf:type :Encompassed;
       :percent 100 ;
       :encompassedArea <http://www.semwebtech.org/mondial/10/country/CZ/> ;
       :encompassedBy <http://www.semwebtech.org/mondial/10/continent/Europe/>] .
<http://www.semwebtech.org/mondial/10/country/CZ/> :encompassed <http://www.semwebtech.org/mondial/10/continent/Europe/> .

 [ rdf:type :Encompassed;
       :percent 100 ;
       :encompassedArea <http://www.semwebtech.org/mondial/10/country/D/> ;
       :encompassedBy <http://www.semwebtech.org/mondial/10/continent/Europe/>] .
<http://www.semwebtech.org/mondial/10/country/D/> :encompassed <http://www.semwebtech.org/mondial/10/continent/Europe/> .
```

Figure 2.20: Example data item using the *Encompassed* class

2.2 RelationalModel & connected Tools

This section will discuss the RELATIONALMODEL, which is our internal model for storing the OBDA mappings and is the common foundation for our developed tools including the RELMODELBUILDER. It will start off with the RDF2SQL tool, which sparked the construction of the RELATIONALMODEL and then explain the components of the RELATIONALMODEL. After that an outlook to the SCHEMAMATCHER tool will be given that is currently in development and will also be based on the RELATIONALMODEL.

2.2.1 RDF2SQL

As mentioned in Section 2.1.3 it is quite difficult to manage the triple data from RDF data sets in a relational database. This is why we previously developed a tool named RDF2SQL consisting of two parts, the DATABASECONVERTER [RS13] and the QUERYCONVERTER [RS14].

The DATABASECONVERTER automatically constructs a relational database based on a given RDF ontology through the steps detailed in [HM12]. The general process on how RDF classes and properties are mapped to a relational database are illustrated in Section 3.1. In this sequence the converter also creates various metadata tables summarised in Section 2.2.2 that store the details of the mapping for further use. As mentioned before these metadata tables serve as a common pivotal point for the remainder of our tools. For example the QUERYCONVERTER of RDF2SQL uses those metadata tables to map SPARQL-queries directed to the original RDF ontology to SQL-queries for the converted relational database. Thus with this tool it is possible to combine the highly variable query language SPARQL with the efficient data storage of a relational database.

2.2.2 RelationalModel

The RELATIONALMODEL is a Java class that stores for an RDF ontology all relevant information to map the ontology to a relational schema, thus providing OBDA to the relational database. This does not include detailed information of the ontology itself, but a structure of *Table* objects representing the mapped relational tables and a series of metadata tables that store the mappings from the RDF classes to the tables. The entirety of the metadata tables is referred to as the *RelationalModel*, because the Java object functions more like an internal representation that can be used to set up the mapping and then store the tables to the database. All other tools accessing the metadata will predominantly do so by querying the metadata tables in the database to keep tools creating the *RelationalModel* independent from those using it. To avoid further confusion between the *RelationalModel* metadata tables and the actual RM, we will refer to the metadata tables as the "inter-model mapping metadata" for the rest of the thesis.

Following are the description of all metadata tables:

MappingDictionary

The *MappingDictionary* is the most important table of the metadata tables. It stores the mapping of class & property pairs to the relational tables:

$$(Class, Property) \mapsto (Table, Column)$$

Additionally the table stores the (abstract) range of the property and if it is stored inversely or a view property. Overall the table is implemented as pictured in Figure 2.21.

MappingDictionary						
class	property	range	tablename	lookupattr	inv	isview

Figure 2.21: MappingDictionary metadata table

- Class: The class to which the property belongs
- Property: The property that is mapped to a relational table
- Range: The abstract range of the property inside of the database, e.g. the literal-valued property range "string" is transformed into "VarChar(199)"
- Tablename: The name of the table the property is mapped to
- Lookupattr: The column name where the data of the property is stored in the table
- Inv: True or False depending on if the property is stored inversely

- IsView: True or False depending on if the property is used as a view over ReifiedTables. This will be futher explained in Sections 3.1.3 and 3.2.1.5.

Using previous examples the property *capital* from class *country* is mapped to the column *capital* in table *Country*, described by the following entries:

class	property	range	tablename	lookupattr	inv	isview
country	capital	VARCHAR	Country	capital	false	false
city	isCapitalOf	VARCHAR	Country	capital	true	false

AllCl

The *AllCl* table stores all $(SubClass, SuperClass)$ relations that are defined in the RDF ontology in a two column table as shown in Figure 2.22.

AllCL	
subclass	superclass
River	Water
Mountain	Place
Place	Location

Figure 2.22: AllCl metadata table

This table includes all classes even if they are abstract classes as mentioned in Section 2.1.4. For example the data entries $(River, Water)$ as well as $(Mountain, Place)$ and $(Place, Location)$.

SubCl

Like the *AllCl* table, the *SubCl* table also stores $(SubClass, SuperClass)$ relations. But the data entries are limited to tuples where both *subclass* and *superclass* are considered a concrete class. This table especially helps with distinguishing concrete and abstract classes. because both are used in different ways in the internal algorithms.

Hometables

This optional metadata table stores $(Class, Table)$ tuples, where *class* entries are concrete classes and *table* entries their corresponding *classtable*. Due to the mapping algorithm that is used in RDF2SQL all properties of a concrete classes are generally mapped to the same table, which acts as the so called *hometable* of the class. This helps with keeping the functional properties grouped to their respective class instead of creating one table per property, which would also be possible. Multivalued properties cannot be stored in the hometable, thus are stored in N:M tables. Most of the time the hometable of a class is named after the class, which would make the table redundant. The table is relevant for example when renaming classes caused by the schema mapping of the SCHEMAMATCHER.

Inv

The *Inv* table stores for every object-valued property the respective inverse property. If the ontology does not define an inverse for a given object-valued property, the inter-model mapping metadata creation tools will automatically define one instead. The automatically created inverses are named like the original property with suffix "_Inv". So for example the inverse of property "hasHeadq" will be named "hasHeadq_Inv". The entries in the *Inv* table are added symmetrically for easier lookup, thus creating the entries *(city, city_Inv)* as well as *(city_Inv, city)*.

PropTableMap

The *PropTableMap* table stores in $(property, tablename)$ pairs which properties are mapped to each hometable. This simple auxilary table helps with accessing basic information about tables and check for equally named properties.

NMTables

As mentioned in Section 2.1.1 there are different types of tables in a relational database. The ClassTables store the functional properties belonging to a specific class, while the NMTables store properties describing N:M relationships between ClassTables, which can not be included in the ClassTables. To differentiate both types of tables after the mapping, the *NMTables* metadata table stores the names of alle NMTables. This includes ReifiedTables, which are further explained in Section 3.1.

NMJ

The *N:M Joins (NMJ)* table stores detailed information about NMTables. An NMTable relates two *sides* with each side consisting of one or more classes, which are mapped to ClassTables. The NMTable connects to each ClassTable through a FK. When looking up the property that is mapped to the NMTable for one class, the connected other FK has to be returned. This information is especially important for generating join-conditions in the QUERYCONVERTER tool.

The *NMJ* table stores for each class on one side their own FK and the FK of the respective other side. Thus the table contains four columns $(class, tablename, lookupattr, fkjoinattr)$ as shown in Figure 2.23.

NMJ			
class	tablename	lookupattr	fkjoinattr

Figure 2.23: NMJ metadata table

- Class: The name of the class of one side
- Tablename: The name of the NMTable
- Lookupattr: The column belonging to the other sides FK
- FkJoinAttr: The column belonging to this class' FK, which have to be joined with the PK of the respective ClassTable

One example of such a property from the RDF ontology is the "locatedAt" property that relates *Cities* to all *Water* subclasses. *Water* is an abstract class combining the classes *Sea*, *River* and *Lake*. Thus one side is the *City* class and the other side is compound of *Sea*, *River* and *Sea*. Assuming the PKs of all three *Water* tables are URI columns, it is possible to combine them into one FK column in the "locatedAt" table, resulting in the columns $(City, Water)$. This results in the four NMJ entries:

class	tablename	lookupattr	fkjoinattr
City	locatedAt	Water	City
Sea	locatedAt	City	Water
Lake	locatedAt	City	Water
River	locatedAt	City	Water

AbstractSubclCols

The *AbstractSubclCols* table stores subclasses that were identified inside of ClassTables. The method to identify those subclasses is explained in Section 3.4.1. Generally the subclasses are identified by one specific value in a property/column that makes up a big portion in that column. The table consists of five columns as illustrated in Figure 2.24.

AbstractSubclCols				
classname	property	localclassname	columnname	isclassvalue

Figure 2.24: AbstractSubclCols metadata table

- Classname: The name of the new subclass that was identified
- Property: The property that is used to define the subclass
- LocalClassname: The class which is mapped to the ClassTable where the subclass was identified
- Columnname: The column in the ClassTable that is used to identify the subclass

- IsClassValue: The value in the the column that identifies the subclass

The *mountain* class for example has a property *type* which specialises the mountain entries into different types. Some mountains with a detailed type are "volcano"s, which offers itself as a good subclass of mountain. The *AbstractSubclCols* entry looks like this:

classname	property	localclassname	columnname	isclassvalue
volcano	type	mountain	type	volcano

Keys

The *Keys* table illustrated in Figure 2.25 is the newest addition to the metadata tables and is especially important when working with relational databases that are not based on URIs. As the name suggests it stores all PK and FK information that can be gathered from the tables. So unlike with URIs when PKs can consist of multipe columns, it is important to distinguish which column of a FK references which column in the respective PK. Additionally due to the fact that classes and properties can be named differently from their mapped counterparts, they have to be stored separately resulting in a total of seven columns:

Keys						
classname	propertyname	tablename	columnname	refclassname	reftablename	refcolumnname

Figure 2.25: Keys metadata table

- Classname: The class to which the property belongs to
- Property: The property or keyword that belongs to the column
- Tablename: The table from which the key is
- Columnname: The column that is constrained by the key
- RefClassname: If the entry is for a FK: the class, which is represented by the referenced PK table; Otherwise: NULL
- RefTablename: If the entry is for a FK: the table that the referenced PK belongs to; Otherwise: NULL
- RefColumnname: If the entry is for a FK: the column in the PK that is referenced by this specific column of the FK; Otherwise: NULL

The table entries are structured based on their specialcase and are further explained in Section 3.2. For example the last three columns are only filled if the table entry is about a FK like the "country" column in the "city" table that references the "code" column in the "country" table:

classname	propertyname	tablename	columnname	refclassname	reftablename	refcolumnname
city	country	city	country	country	country	code

If the FK consists of multiple column, each column would have an individual entry.

2.2.3 SchemaMatcher

The SCHEMAMATCHER [Sch16] tool is a work-in-progress project that will advance the possibilities of our OBDA mappings to be used for converging similar relational databases. The functionality of the SCHEMAMATCHER is highlighted in Figure 2.26 with the green dots for relational databases representing *tables* and *attributes* and the red dots for ontological vocabularies representing *classes* and *properties*. First of all the RELMODELBUILDER will be used to extract individual ontological metadata and the respective mappings from each relational database. The SCHEMAMATCHER is then utilised to measure the similarity of the different extracted ontological vocabularies and the structures of the relational databases. After that the similarity score is used to align the ontological vocabularies and create mappings of classes and properties from one database to the other.

These mappings will serve multiple purposes:

1. They can be used to guide translation-steps to try to adapt one database to the other with the help of atomic schema operators
2. The mappings can be used by the QUERYCONVERTER tool to construct SQL queries for both databases from SPARQL queries based on the shared ontological vocabulary

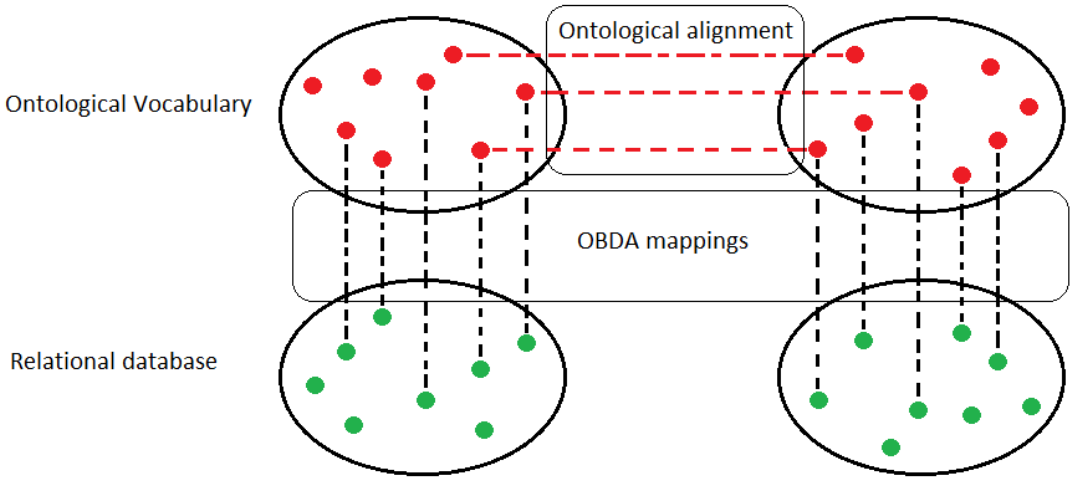


Figure 2.26: Interaction of the RELMODELBUILDER with the SCHEMAMATCHER

Chapter 3

Connecting RDF model and relational model

This chapter discusses the theoretical foundation of the thesis. For example the mapping of tables to classes, restrictions of the relational model compared to the RDF, and identifying different types of tables and their connections to each other. First of all, the basic principles about the foundation given from the previous work will be reviewed, which maps an RDF model to relational tables. After that the differences and special cases that emerge when constructing the backwards mapping are discussed. Examples are given from the previously discussed MONDIAL database introduced in Section 1.5.

3.1 Mapping the RDF model to relational tables

This section explains how data according to the RDF/OWL ontology is mapped to the relational model. At first the general rules for mapping concrete classes and their functional properties are stated. Then the differences for non-functional properties and the special case of reified properties will be discussed. More detailed information on how to map an OWL ontology to relational tables can be found in [HM12]. All examples given for RDF/OWL ontology definitions are excerpts of the MONDIAL ontology with possibly slight deviation for easier explanation. The full definitions can be found in the Appendix A and B.

3.1.1 Mapping of concrete classes and functional properties

In the RDF/OWL model the user can define *classes* to distinguish resources and assign *properties* to such classes, which can be used in triples to assign property specific values to each resource as explained in Section 2.1.3. This makes it easy for querying to address only a group of data items. In a relational database a similar classification exists in the form of *relations*. A relation defines a group of data items that have the same *attributes*. Therefore for a forward mapping from RDF to the relational model it can be assumed in general that concrete OWL *classes* are mapped to

relational model *relations* and their respective *functional properties* to *attributes*. These relations are then saved as *tables* in a relational DBMS and attributes make up the *columns* of the table.

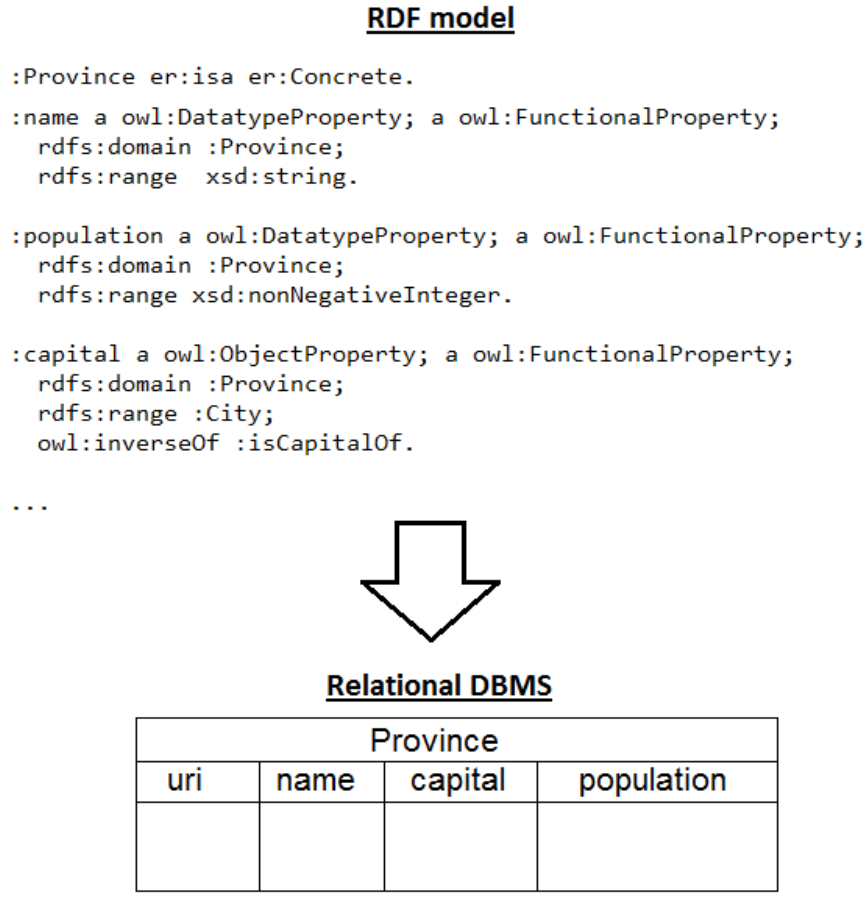


Figure 3.1: Mapping of classes and functional properties to tables

Figure 3.1 shows for instance the mapping of class *Province* and some of its functional properties to the relational database. *Province* is defined as a concrete class and has among others the functional properties *(name, capital, population)*. First of all a "uri" column has to be added that functions as the PK of the table and stores the URIs of each instance of the *Province* class. The functional properties are then also directly mapped to columns of the table "Province". The datatype of the columns is dependent on the range and type of the properties. For instance in an ORACLE DBMS the column "name" for the *DatatypeProperty name* with range *xsd:string* gets the datatype *VarChar*, while the *population* property with range *xsd:nonNegativeInteger* gets the datatype *DECIMAL*. The "uri" column gets the datatype *Varchar*, because the URIs of RDF resources are defined to be strings. Thus, object-valued properties like *capital* also always get the datatype *VarChar*, because they reference to the "uri" column of another table. This reference to the concrete *City* class can also be further specified by adding a FK constraint to the column. In

the end every instance of the *Province* class is then represented as one data entry in the table.

In the actual ontology these properties would not have only the *Province* class as their domain, but an abstract superclass that describes all suitable users of the property. For instance the *population* property should be usable by every subclass of the *Area* abstract superclass which includes for example the *City* and *Country* classes, too. Abstract classes however are not mapped to tables in the relational database.

3.1.2 Mapping of non-functional properties

Such a mapping is not as straightforward for *non-functional properties*, because one instance of a class can have a varying amount of these properties making them difficult to implement in the mapped table of the class. Thus those properties are mapped to an additional table with an N:M connection to the table of the source class (also called home table).

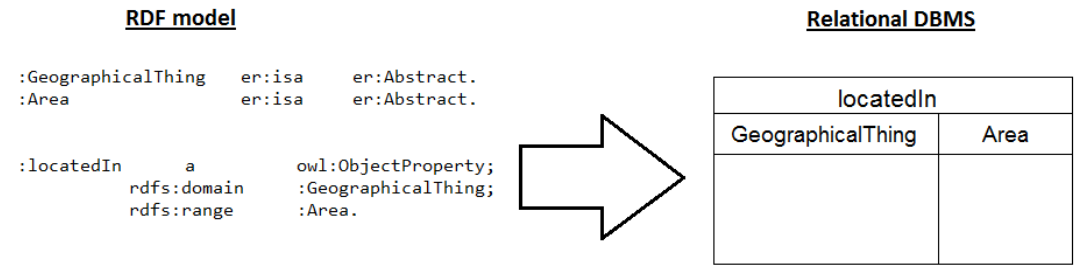


Figure 3.2: Mapping of non-functional properties to N:M tables

Figure 3.2 shows for instance such a mapping for the property *locatedIn*. The *locatedIn* property is of type *ObjectProperty* with domain *GeographicalThing* and range *Area*. Both *GeographicalThing* and *Area* are abstract classes that each cover various concrete subclasses. For instance the classes *Mountain* and *River* are concrete subclasses of *GeographicalThing*, while the *Area* superclass contains amongst others the concrete classes *Province* and *Country* as mentioned in Section 3.1.1. Thus, this property is mapped to an equally named table "locatedIn" with column "GeographicalThing" and "Area", which stores for example $(Mountain, Country)$ tuples relating mountains and the respective country (or countries) they are located in. Being object-valued both columns get the datatype *VarChar*, but due to the fact that each column references multiple different tables no FK constraint can be defined for either of them.

3.1.3 Reified properties and their mapping

As already mentioned in Section 2.1.4.1, there is no official standard for defining reified properties in an OWL ontology. However when simulating reified properties in RDF/OWL ontologies with our annotations, they can be mapped to the relational database, too. Figure 3.3 shows again the *encompassed* property example that is reified to the concrete class *Encompassed* with

the functional properties *encompassedArea*, *encompassedBy* and *percent*. As such the mapping of the reified property *Encompassed* and its functional properties can follow the mapping schema introduced in Section 3.1.1. The original property *encompassed* is not mapped to the relational database.

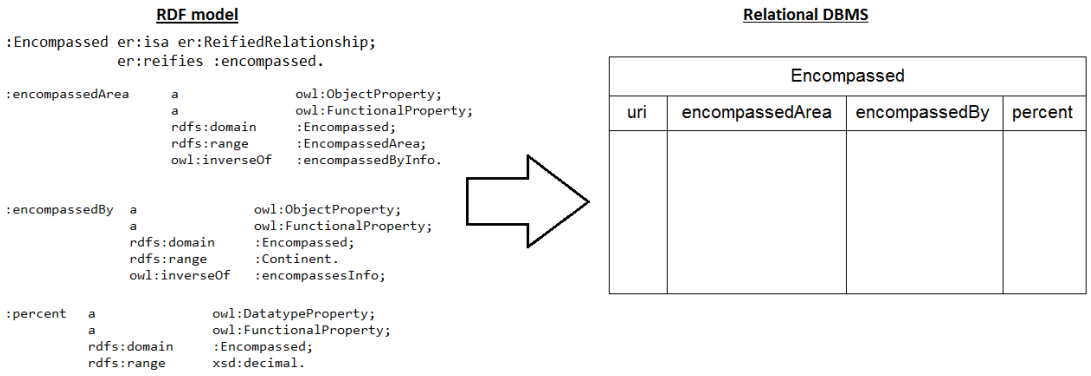


Figure 3.3: Mapping of reified properties to individual tables

Encompassed is mapped to a table in the relational database with a "uri" column even though the *Encompassed* resources in the MONDIAL RDF database are always used with blank nodes. The values in this column are then the internal unique identifiers used in the SPARQL query-engine that is used to access the data to transfer the data from theRDF database to the relational database. The three functional properties are then added as columns with their respective datatype.

3.1.4 Mapping of symmetric properties

Symmetric properties are a special case of recursive properties. For a recursive property the domain and the range has to be the same, while additionally for a symmetric property must hold:

$$symmproperty(A, B) \iff symmproperty(B, A)$$

An example of a symmetric property is the *mergesWith* property that relates two seas that merge together. Obviously if sea A merges with sea B, the same applies the other way around. The definition and mapping of the *mergesWith* property is shown in Figure 3.4. As a non-functional property it has to be mapped to an N:M table with the name "mergesWith". However both the domain-side and the range-side are the same resulting in the issue that both columns should normally be named "Sea". This is not possible in a relational database, thus the columns have to be distinguished by naming them "Sea1" and "Sea2". Regardless of the naming the OBDA mapping stays intact, because our inter-model mapping metadata supports the renaming of columns.

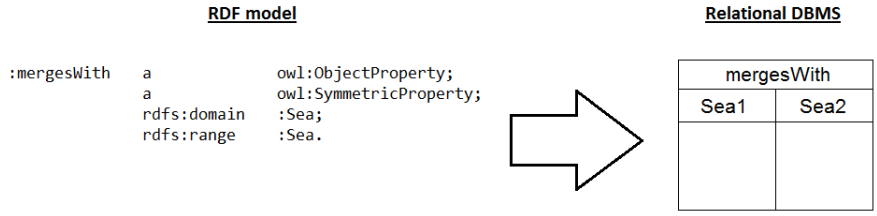


Figure 3.4: Mapping of a symmetric property to the relatiotional database

On the other hand. the mapping of functional properties is not affected by the symmetry. It is however also possible that a symmetric property is reified to a class, which we will further call *symmetric reified* (or short: *symmreified*) properties.

The *neighbor* property with domain and range of the *Country* class is an example for such a symmetric property that is reified in the MONDIAL OWL ontology. The definition of the *neighbor* property is illustrated in Figure 3.5. As in the previous case with the *encompassed* property explained in Section 2.1.4.1, reification is only necessary if additional properties should be linked to the *neighbor* property. In this case a *length* property is added that describes the length of the connecting border. Thus, a new *Border* class is defined in the MONDIAL OWL ontology that reifies the *neighbor* property. The definition of the *Border* class and its *bordering* property with the resulting mapping to a relational table is shown in Figure 3.6.

```
:neighbor  a                owl:ObjectProperty;
           a                owl:SymmetricProperty;
           rdfs:domain      :Country;
           rdfs:range       :Country.
```

Figure 3.5: Definition of the *neighbor* property

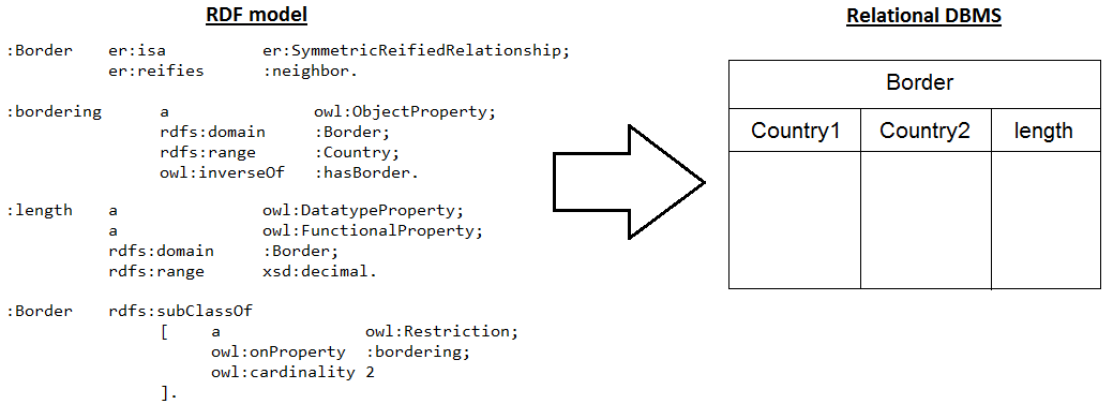


Figure 3.6: Mapping of symmreified properties to individual tables

The *Border* class is defined as an instance of the *SymmetricReifiedRelationship* class, which is a subclass of the previously used *ReifiedRelationship* class. For normal reified properties two new properties would be defined now, one for the original property's domain-side and one for the

range-side. However due to the symmetry of the *neighbor* property, no roles are defined for either side and thus only one new property named *bordering* is enough that is used two times. This characteristic is defined by the *owl:Restriction* restraining the *bordering* property to a cardinality of "2", which means that the *bordering* property has to be used exactly two times for each instance of *Border*.

As with normal symmetric properties, when mapping *Border* to a table in the relational model with the approach described in Section 3.1.3, it would be necessary to add two columns with the same name "bordering", which is not possible in the relational model. Therefore the naming of the columns is oriented to the naming approach of mapping normal symmetric properties. Thus, the range *Country* of the *bordering* property is taken as the base and then distinguished by adding the enumeration. Finally the *length* property is treated like the *percent* property for the *Encompassed* class.

3.2 Extracting an RDF model from relational tables

After reviewing the basics of the forward mapping from a RDF model to relational tables, this section covers the backwards mapping of extracting classes and properties from a given relational schema. The objective of this process is not only to define the raw classes, but also to fill the metadata tables described in Section 2.2.2. Therefore the first section is split into parts, each examining the mapping of a different kind of table and what metadata table entries are created with the gained information. Then we will talk about why we chose the used naming conventions and why only a limited class hierarchy can be derived from the information. In the end, an outlook will be given on how to identify ranges of properties if not all foreign keys are given.

3.2.1 Mapping of tables

Unfortunately the backwards mapping from relational tables to RDF classes is not as clear as the forward mapping. The main reason for this is that the tables are not as clearly distinguished as the diverse kinds of classes and properties in the RDF model. Therefore it is essential to first classify the given tables and then map them according to the identified type. To better discuss the different types of tables, they are named according to the conceptual terms of the RDF model they are representing:

- ClassTables: Represent classes of the OWL model or entity types of the **ERM!** (**ERM!**)
- NMTables: Represent non-functional properties of the RDF/ER-model describing N:M relationships between classes.
- ReifiedTables: Represent reified non-functional properties of the RDF/ER-model. They are a sub-type of ClassTables.

Due to lack of metadata information about the tables the classification takes primarily the primary and foreign keys of a table into account. Thus, for the presented mapping it is presumed that all primary and foreign keys of the relational model are given.

3.2.1.1 Standard ClassTables

As described in the forward mapping not every table represents a class. But due to the fact that *ReifiedTables* are more akin to *ClassTables* than to *NMTables*, and infact are ClassTables representing the reified relationships, it is highly likely that every table except pure N:M tables are *ClassTables* to begin with. They can then be specialized later on.

To explain the mapping of standard ClassTables a closer look at the "country" table will be taken. The definition of the table taken from the relational version of the MONDIAL database in POSTGRESQL is shown in Figure 3.7 and an excerpt of the data in the table can be found in Figure 3.8. As the create table statement shows the "country" table has several attributes and constraints. For the classification of the table only the primary key and foreign key constraints are of importance, but the remaining constraints could be used if a more comprehensive ontology should be constructed. The primary key of the table is the column "code" and there exists a foreign key to the capital of the country in the "city" table. A city is uniquely identified by its name and the names of the country and province it is located in.

```
CREATE TABLE country
(
  name character varying(50) NOT NULL,
  code character varying(4) NOT NULL,
  capital character varying(50),
  province character varying(50),
  area numeric,
  population numeric,
  CONSTRAINT countrykey PRIMARY KEY (code),
  CONSTRAINT countryrefscap FOREIGN KEY (capital, code, province)
     REFERENCES city (name, country, province) MATCH SIMPLE
     ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT country_name_key UNIQUE (name),
  CONSTRAINT countrynameunique UNIQUE (name),
  CONSTRAINT countryarea CHECK (area >= 0::numeric),
  CONSTRAINT countrypop CHECK (population >= 0::numeric)
);
```

Figure 3.7: Create table statement of the "country" table of the relational version of the MONDIAL database

	name character varying(50)	code [PK] character varying(4)	capital character varying(50)	province character varying(50)	area numeric	population numeric
1	Austria	A	Wien	Wien	83850	8499759
2	Afghanistan	AFG	Kabul	Afghanistan	647500	26023100
3	Antigua and Barbuda	AG	Saint Johns	Antigua and Barbuda	442	81799
4	Albania	AL	Tirana	Albania	28750	2800138
5	American Samoa	AMSA	Pago Pago	American Samoa	199	55519
6	Andorra	AND	Andorra la Vella	Andorra	450	78115
7	Angola	ANG	Luanda	Luanda	1246700	24383301
8	Armenia	ARM	Yerevan	Armenia	29800	3026879
9	Aruba	ARU	Oranjestad	Aruba	193	101484
10	Australia	AUS	Canberra	Australia Capital Territory	7686850	23135281

Figure 3.8: Excerpt of the data in the "Country" table

Being a ClassTable it can be defined that the table "country" maps to an equally named class *Country*. Additionally, the class has the functional properties *name*, *code*, *capital*, *province*, *area* and *population*. As such it is known that these functional properties can be found in the table "country" thus the following entries are added to our MappingDictionary:

MD						
class	property	range	tablename	lookupattr	inv	isview
country	name	VARCHAR(199)	country	name	false	false
country	code	VARCHAR(199)	country	code	false	false
country	capital	VARCHAR(199)	country	capital	false	false
country	province	VARCHAR(199)	country	province	false	false
country	area	DECIMAL	country	area	false	false
country	population	DECIMAL	country	population	false	false

From the constraints of the table it can be identified that the columns "capital", "code" and "province" form together a FK to the "city" table. This concludes that the three columns represent a single object-valued property with domain *country* and range *city*. The discrepancy that three columns are mapped to a single property arises, because in the RDF model the instances of a class are always uniquely addressable through their URIs, but in the relational model the PK can contain multiple columns. Thus, an auxiliary property has to be created named after the columns in the FK *capital_code_province* that represents this relationship. Being object-valued also means that there theoretically exists an inverse property for it named *capital_code_province_Inv*, therefore these entries are added in the *Inv* table:

Inv	
property	inverse
capital_code_province	capital_code_province_Inv
capital_code_province_Inv	capital_code_province

Both new properties also need to be added to the *MD* so they can be used in SPARQL queries for the QUERYCONVERTER tool:

MD						
class	property	range	tablename	lookupattr	inv	isview
country	capital_code_province	VARCHAR(199)	country	capital_code_province	false	false
city	capital_code_province_Inv	VARCHAR(199)	country	capital_code_province	true	false

The inverse property points from the opposite direction thus the "class" needs to be switched to *city* and the "inv" attribute set to *true*.

Finally the PK and FK can be added to the *Keys* table. For the PKs we decided to resuse the keyword "uri" as the property name taken from the ontology-to-relational mappings. The actual property name for PKs does not matter in the *Keys* table and the usage of such a keywords simplifies the filtering process for the other tools that access this table. If PK consists of multiple columns, each gets its own entry. Similar to this are the entries for the multi-column FKs, where one table entry for each column involved in the FK is necessary, because each column targets one specific column in the referenced table. The FK columns are grouped together through the same propertyname that describes this FK. All in all the following entries are created:

Keys						
classname	propertyname	tablename	columnname	refclassname	reftablename	refcolumnname
country	uri	country	code			
country	capital_code_province	country	capital	city	city	name
country	capital_code_province	country	province	city	city	province
country	capital_code_province	country	code	city	city	country
city	uri	city	name			
city	uri	city	province			
city	uri	city	country			

3.2.1.2 Special case ClassTableExtensions

Unlike the mapping from the RDF model to relational tables there is a special case regarding ClassTables for the mapping in the opposite direction. Often in practice when creating relational databases the user splits a otherwise singular table into several by vertical partitioning. This can be done due to several reasons. For example the creator wants to avoid a single table with many columns for efficiency reasons or just wants to group the data for an entity depending on the topic to make the data more human readable.

The "country" table can be taken as an example whose columns were already covered previously. However the MONDIAL database covers not only those basic information about countries, but for instance also information regarding the economy of the country. This information is not directly stored in the "country" table to avoid bloating it, but was outsourced to the "economy" table, which is linked to the "country" table through a FK. The definition of the "economy" table is illustrated in Figure 3.9. The constraints show that the "country" column is both the PK of the "economy" table and the FK to the "country" table. This implies an 1:1-relationship between "country" and "economy" and thus the columns of "economy" could have been stored in the "country" table. Such a table would not have been created, when using the RDF2SQL tool to map an RDF ontology to relational tables, it would have created a single, broad table instead.

```
CREATE TABLE economy
(
  country character varying(4) NOT NULL,
  gdp numeric,
  agriculture numeric,
  service numeric,
  industry numeric,
  inflation numeric,
  unemployment numeric,
  CONSTRAINT economykey PRIMARY KEY (country),
  CONSTRAINT economyref FOREIGN KEY (country)
     REFERENCES country (code) MATCH SIMPLE
     ON UPDATE NO ACTION ON DELETE CASCADE,
  CONSTRAINT economygdp CHECK (gdp >= 0::numeric)
);
```

Figure 3.9: Create table statement of the "economy" table of the relational version of the MONDIAL database

Therefore for the relational schema to RDF ontology mapping, we define a table to be a so called *ClassTableExtension* when:

1. The PK of the table is the FK to a ClassTable and
2. There is no other table with a FK that references this table

The main ClassTable will be referenced at least from the ClassTableExtension, so it is important that there exists no symmetric reference. This will get problematic, when not all FKs are given and the RELMODELBUILDER has to identify them on its own as described in Section 3.5.

A ClassTableExtension and its columns will not be mapped to an individual class, but to the class that is represented by the ClassTable it is the extension of. In our example this would be the class *country*. Thus the following *MappingDictionary* entries are added:

MD						
class	property	range	tablename	lookupattr	inv	isview
country	gdp	DECIMAL	economy	gdp	false	false
country	agriculture	DECIMAL	economy	agriculture	false	false
country	service	DECIMAL	economy	service	false	false
country	industry	DECIMAL	economy	industry	false	false
country	inflation	DECIMAL	economy	inflation	false	false
country	unemployment	DECIMAL	economy	unemployment	false	false

The "country" column of the table does not need to be mapped to a property, because it is just used as a connection means by the relational database. This also means that no inverse of the otherwise object-valued property has to be created. However an entry for the FK has to be added to the *Keys* table, so that the QUERYCONVERTER tool is able to join the "country" and "economy" table if necessary. Because no property is created for this column, the keyword "extension" is used for easier identification:

Keys						
classname	propertyname	tablename	columnname	refclassname	reftablename	refcolumnname
country	extension	economy	country	country	country	code

The entry for the PK is also omitted, because the hometable of the *country* class is the "country" table.

Some other tables may at first glance seem to be ClassTableExtensions too, for example the "countrypops" table. This table contains the columns $(country, year, population)$ with column "country" being a FK to the respective "country" table. However, the PK of the table is the tuple $(country, year)$ indicating that this table does not have a 1:1 relationship to the "country" table but an N:1 relationship. The table contains multiple population entries for each country depending on the year and thus is no simple extension of the original ClassTable. The table has to be mapped to the distinct reification *countrypops*.

3.2.1.3 NMTables

As mentioned before, we distinguish two *sides* for each NMTable, where one side is called the *domain side* and the other the *range side*. Because there is no meta information about abstract superclasses in the relational tables most sides will only have one class/table. The FKs of both sides have to involve all columns of the table together and thus, there are no additional columns. A PK is optional, but if present it contains all columns in the table. Additionally because of that, NMTables should not be referenced to.

An example for such a pure NMTable is the "locatedon" table, which stores which cities are located on which islands. The definition of the table is shown in Figure 3.10. The table has a PK that covers all four columns and two FKs that also cover all columns together, but have distinct differences. On the domain side, the FK consisting of the columns $(city, country, province)$ references the "city" table and on the range side the "island" column references the "island" table. Both referenced tables are ClassTables and the "locatedon" table is not referenced from another table.

```
CREATE TABLE locatedon
(
 city character varying(50) NOT NULL,
 province character varying(50) NOT NULL,
 country character varying(4) NOT NULL,
 island character varying(50) NOT NULL,
 CONSTRAINT locatedonkey PRIMARY KEY (city, province, country, island),
 CONSTRAINT locatedonrefscity FOREIGN KEY (city, country, province)
    REFERENCES city (name, country, province) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE CASCADE,
 CONSTRAINT locatedonrefsisland FOREIGN KEY (island)
    REFERENCES island (name) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
);
```

Figure 3.10: Create table statement of the "loactedon" table of the relational version of the MONDIAL database

Now that the "locatedon" table is identified as an NMTable to start creating the metadata table entries, its name is added to the list of *NMTables*:

NMTables
tablename
locatedon

Then the connection entries in the *NMJ* table are added:

NMJ			
class	tablename	lookupattr	fkjoinattr
city	locatedon	island	city_country_province
island	locatedon	city_country_province	island

The first entry means that the "locatedon" table is joined with the hometable of the *city* class from the first side with the columns "city","country" and "province", while the FK to the other sides hometable is the "island" column. The column-names have to be grouped again for the multi-column FK, because multiple entries would hint to a *CompositeNMTable*, which are a special case of NMTables and will be explained in Section 3.2.1.4. Tools that use these entries have to parse the results if necessary. Additionally the property to address this relationship in a SPARQL-query has to be named more detailed as to just the tablename *locatedon*, because the program cannot automatically decide which of the two sides is the domain-side of the property. Thus, the column names as used in the "lookupattr" of the opposite side is simply added to the property for each class to give it directional context. This results in the property *locatedon_island* for the class *city* and the property *locatedon_city_country_province* for the class *island*. It is not necessary to add these suffixes if the NMTable represents a symmetric property, because then the direction is irrelevant. Regardless of the naming the properties are inverse to each other and thus have to be added to the *Inv* table:

Inv	
property	inverse
locatedon_island	locatedon_city_country_province
locatedon_city_country_province	locatedon_island

After defining the attribute and the property names, the respective *MD* entries are added:

MD						
class	property	range	tablename	lookupattr	inv	isview
city	locatedon_island	VARCHAR(199)	locatedon	island	false	false
island	locatedon_city_country_province	VARCHAR(199)	locatedon	city_country_province	false	false

On the other hand, the entries for NMTables in the *Keys* table are not as straightforward. No entries for the PK have to be added, because an NMTable is not mapped to an individual class. Furthermore because of that the entries for the FKs do not have a class or property associated with them. The previously defined properties always point towards the FK of the opposite side for the N:M relationship, not to the FK pointing to its own hometable. This results in the following entries:

Keys						
classname	propertyname	tablename	columnname	refclassname	reftablename	refcolumnname
		locatedon	city	city	city	name
		locatedon	country	city	city	country
		locatedon	province	city	city	province
		locatedon	island	island	island	name

3.2.1.4 Special case CompositeNMTables

Like with ClassTables we found a special case for NMTables, when mapping in the relational table to ontology direction. The point of interest is the "located" table, which stores for every city at which river, sea or lake it is located. As mentioned in Section 2.2.2 the classes *river*, *sea* and *lake* are grouped in the abstract superclass *Water* in the RDF ontology. This superclass could be used in the ontology-to-relational-model-mapping to define a "locatedAt" NMTable, which consists of only two columns "city" and "water". The column "water" then contained the entries for all three concrete subclasses.

This setup is not possible starting from the relational database, because one column can not have three FK constraints at the same time. If FK constraints should be used in the database, the "water" column has to be split into three seperate columns "sea", "lake" and "river". Furthermore unchanged from previous examples if no URIs are used the FK referencing to the "city" table has to have three columns, too. The table definition and an illustration of the table structure is shown in Figure 3.11.

```
CREATE TABLE located
(
  city character varying(50),
  province character varying(50),
  country character varying(4),
  river character varying(50),
  lake character varying(50),
  sea character varying(50),
  CONSTRAINT locatedlake FOREIGN KEY (lake)
     REFERENCES lake (name) MATCH SIMPLE
     ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT locatedrefscity FOREIGN KEY (city, country, province)
     REFERENCES city (name, country, province) MATCH SIMPLE
     ON UPDATE NO ACTION ON DELETE CASCADE,
  CONSTRAINT locatedriver FOREIGN KEY (river)
     REFERENCES river (name) MATCH SIMPLE
     ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT locatedsea FOREIGN KEY (sea)
     REFERENCES sea (name) MATCH SIMPLE
     ON UPDATE NO ACTION ON DELETE NO ACTION
);
```

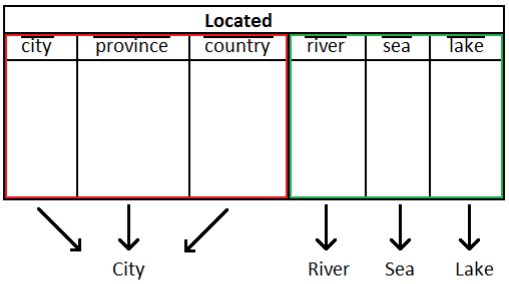


Figure 3.11: Structure of the "located" table from the relational MONDIAL database

The goal for these tables is to identify a *dominant* side that has a relationship with each of the remaining FKs. This can be done through analysing the data entries of the table, especially the occurence of NULL-values and is further explained in Section 4.3.4. In this case the dominant side of the table is the FK to the "City" table consisting of the columns $(city, province, country)$ marked in red. Thus the dominant side stands in a relationship with the remaining FKs $(river)$, (sea) and $(lake)$. Therefore using the previous approach for NMTables, for each such relationsip two entries are needed in the *NMJ* table, one for each direction of the relationship:

NMJ			
class	tablename	lookupattr	fkjoinattr
city	located	river	city_country_province
city	located	sea	city_country_province
city	located	lake	city_country_province
sea	located	city_country_province	sea
lake	located	city_country_province	lake
river	located	city_country_province	river

Following the ontology functionality that *river*, *sea* and *lake* are grouped, the *city* class needs one property to access all three of them together. For this the three separate FKs are treated as if they are combined. Thus the property will be named *located_lake_river_sea* and the inverse from the direction of the opposite side *located_city_country_province*. These properties can then be used in the *MD* entries for each class in their respective side:

Inv	
property	inverse
located_lake_river_sea	located_city_country_province
located_city_country_province	located_lake_river_sea

MD						
class	property	range	tablename	lookupattr	inv	isview
city	located_lake_river_sea	VARCHAR(199)	located	lake_river_sea	false	false
river	located_city_country_province	VARCHAR(199)	located	city_country_province	false	false
lake	located_city_country_province	VARCHAR(199)	located	city_country_province	false	false
sea	located_city_country_province	VARCHAR(199)	located	city_country_province	false	false

The entries up until now are constructed as if the table were a normal NMTable with *city* being the domain-side and (*river*,*lake*,*sea*) grouped as the range-side of the non functional property. These entries are for accessing all three of the relationships at the same time without specifying if *river, lake* or *sea* is needed. Now the property entries have to be added for the specific relationships $(city, river)$, $(city, lake)$ and $(city, sea)$. These properties make it possible in a SPARQL-query to specifically get all cities located at a river and not the other two water types. For better readability, the constructed properties are named in such a way that they reflect the direction inside of the relationsip. For example the property of *city* to *river* is named *located_city_to_river* and the inverse *located_river_to_city*. All in all the following six properties are added to the *Inv* table:

Inv	
property	inverse
located_city_to_lake	located_lake_to_city
located_lake_to_city	located_city_to_lake
located_city_to_river	located_river_to_city
located_river_to_city	located_city_to_river
located_city_to_sea	located_sea_to_city
located_sea_to_city	located_city_to_sea

These properties can then again be used for the *MD* entries:

MD						
class	property	range	tablename	lookupattr	inv	isview
city	located_city_to_lake	VARCHAR(199)	located	lake	false	false
city	located_city_to_river	VARCHAR(199)	located	river	false	false
city	located_city_to_sea	VARCHAR(199)	located	sea	false	false
river	located_river_to_city	VARCHAR(199)	located	city_country_province	false	false
lake	located_lake_to_city	VARCHAR(199)	located	city_country_province	false	false
sea	located_sea_to_city	VARCHAR(199)	located	city_country_province	false	false

Finally the entries for the *Keys* table follow the same schema as a common NMTable, only the amount of FKs is different:

Keys						
classname	propertyname	tablename	columnname	refclassname	reftablename	refcolumnname
		located	lake	lake	lake	name
		located	sea	sea	sea	name
		located	river	river	river	name
		located	city	city	city	name
		located	country	city	city	country
		located	province	city	city	province

3.2.1.5 ReifiedTables

As explained in Section 3.1.3, ReifiedTables are tables, which connect two ClassTables similar to NMTables, but store additionally information. In the relational model such tables are created for N:M relationships with attributes. The simple relationship is reified to an entity and thus needs to be mapped to an individual class.

To identify ReifiedTables without the meta information from the underlying relational model, each ClassTable has to be checked if the columns involved in the PK make up an N:M relationship. If the table contains additional columns besides the ones in the PK, it is not transformed into a simple NMTable, but is marked as a ReifiedTable. One such ReifiedTable is the "encompasses" table, which contains the columns $(country, continent, percentage)$ with PK $(country, continent)$ and two FKs $(country) \rightarrow country$ and$(continent) \rightarrow continent$. The columns of both FKs in the PK are disjoint and form their respective side of the N:M relationship.

The mapping of a ReifiedTable is similar to the one of a normal ClassTable with the exception that the information of the N:M relationship needs to be mapped too. It starts by defining the *Encompasses* class and adding the structural information of the table to the *Keys* table:

Keys						
classname	propertyname	tablename	columnname	refclassname	reftablename	refcolumnname
encompasses	uri	encompasses	country			
encompasses	uri	encompasses	continent			
encompasses	country	encompasses	country	country	country	code
encompasses	continent	encompasses	continent	continent	continent	name

After that the entries for the ClassTable aspect of the table are created. The columns "country" and "continent" being object-valued in a ClassTable means that also inverses for their properties have to be created following the usual procedure:

Inv	
property	inverse
continent	continent_Inv
continent_Inv	continent
country	country_Inv
country_Inv	country

For the *MD* just one entry for each column is added as usual and the inverses are used with their respective classes instead of *encompasses*:

MD						
class	property	range	tablename	lookupattr	inv	isview
encompasses	country	VARCHAR(199)	encompasses	country	false	false
encompasses	continent	VARCHAR(199)	encompasses	continent	false	false
encompasses	percentage	DECIMAL	encompasses	percentage	false	false
continent	continent_Inv	VARCHAR(199)	encompasses	continent	false	false
country	country_Inv	VARCHAR(199)	encompasses	country	false	false

In the end the entries for the N:M relationship portion of the table are added. Even though the table is generally considered a ClassTable, the table name has to be added to the list of *NMTables* and as well as the lookup information to the *NMJ* just like a normal NMTable:

NMTables
tablename
encompasses

NMJ			
class	tablename	lookupattr	fkjoinattr
country	encompasses	continent	country
continent	encompasses	country	continent

Additionally a reified property and its inverse have to be created for the table usually named after the table. However following the naming schema from above, column names have to be added as suffixes to it to avoid overlapping in case of special cases. After all, the property *encompasses_country* is defined for the class *continent* and for the class *country* the property *encompasses_continent*.

Inv	
property	inverse
encompasses_country	encompasses_continent
encompasses_continent	encompasses_country

Concluding the *MD* entry for each side can be added with the exception that both entries are marked as "isview":

MD						
class	property	range	tablename	lookupattr	inv	isview
continent	encompasses_country	VARCHAR(199)	encompasses	country	false	true
country	encompasses_continent	VARCHAR(199)	encompasses	continent	false	true

Normally in a SPARQL query if one sides class has to be related to the opposite sides class, it has to be first related to the *encompasses* class, which in return has to be related to the opposites sides class.

$$\begin{array}{ccc} ?e & a & :encompasses; \\ & :country & ?c; \\ & :continent & ?con. \\ ?c & a & :country; \\ ?con & a & ?continent. \end{array}$$

By using their respective sides view property the classes of each side can also use the "encompasses" table as a view to directly access the opposite side without first using the *encompasses* class. Note that this is only useful if the *percentage* property does not have to be accessed.

$$\begin{array}{ccc} ?c & a & :country; \\ & encompasses-continent & ?con. \\ ?com & a & :continent \end{array}$$

3.3 Naming properties

When constructing the columns from the properties of the ontology the RDF2SQL tool keeps the names the same. The properties are comprehensible named in the context they are used in. Optimally we would want to do the same when defining properties for the columns. Unfortunately this is not feasible, because in practice the columns in relational databases are often named more simplistic than the respective property would have been. This is especially true for columns used in FKs. For example the column used as the FK to the "country" table is more often than not just named "country" regardless of the table it is used in. On the other hand for example the property of the *Encompassed* class which ranges over the *Country* class is named *encompassedAre*a giving it more context. This discrepancy becomes especially apparent when defining inverses. The relational database has no concept of an inverse FK and thus without further information the automated process is forced to define *(captial, captial_Inv)* pairs instead of *(capital, isCaptialOf)*.

The other discrepancies come from the special cases of the relational database detailed above. First of all it is possible that the PK of a ClassTable consists of multiple columns, while a instance of a class in the RDF ontology is always uniquely identified by its URI. Thus a class is related to another by a single property. To simulate this for multi-column FKs a property has to be defined that includes all columns of the FK by concatenating them in alphabethical order. For example the property *capital_code_province* is constructed for the FK of the "country" table to the "city" table.

Furthermore NMTables in relational databases are generally named quite good to describe the relationship of their sides. But the description is often strongly influenced by a direction from one side to another. For example a *city* is *locatedOn* a *island* with "locatedOn" being the name of the NMTable and "city" and "island" its respective sides. On the contrary a *island* is *locatedOn* a *city* would not make much sense. Unfortunately an automated system can not identify which direction of the two is meant by the creator of the table. Therefore to avoid assigning the wrong direction to a side a directional affix has to be added to the table name for defining the property. The directional affix is for each side the column name(s) of their respective opposite side. Thus the property *locatedOn_island* is created for side "city" and *locatedOn_city* for the side "island".

This restriction applies only to NMTables where the two sides are different. If the NMTable stores a symmetric N:M relationship that is both sides target the same ClassTable, no affix has to be added to the properties because the direction is irrelevant. This holds true for example for the "mergesWith" table, which stores pairs of seas that are connected to each other. Both sides of the table target the "sea" table, which enables the usage of the simple *mergesWith* property for one side and the *mergesWith_Inv* inverse property for the other. This also applies to the N:M portion of symmetric ReifiedTables.

3.4 Class hierarchy

A class hierarchy describes which classes are sub- or superclasses of other classes. This information is stored in the *AllCl* and *SubCl* table as explained in Section 2.2.2. While the RDF/**OWl!** (**OWl!**) ontology supports subclasses and the ENHANCED ENTITY-RELATIONSHIP

MODEL adds the functionality of subclasses and superclasses to the original ER, there exists no equivalent concept (e.g. "subtables") in a relational database to directly identify such relationships between tables. Therefore this section starts with identifying subclasses contained inside of ClassTables and then look at possible relationships between tables. But first of all, generic superclasses for all already identified classes have to be added, which includes all ClassTables and ReifiedTables. The generic superclass for all entries in the *AllCl* in the *SubCl* table is named *rdf2sqlTop*. These entries are made just to construct the bare minimum of a class hierarchy. The entries for the *country* class are as demonstrated:

AllCl	
subclass	superclass
country	rdf2sqlTop

SubCl	
subclass	superclass
country	rdf2sqlTop

3.4.1 Identifying subclasses of ClassTables

Identifying subclasses inside of ClassTables cannot be done based on the definition of the table alone. The table entries have to be analysed and clusters of entries have to be searched that share a common feature and are large enough to warrant the definition of a subclass.

For the RDF2SQL tool the class hierarchy is defined in the OWL ontology and subclasses of concrete classes such as $volcano \quad rdfs:subClassOf \quad mountain$ are mapped to the same table as their superclass. The ClassTable of the superclass gets an additional column named "rdfType", which stores the name of the subclass if the data entry is an instance of that subclass. However starting from the relational table these clusters can depend on multiple columns instead of just one, but to hold the computing time low we decided to limit the search to clusters defined by a single column. To be significant, such *type* columns should not contain more than five different values including the *NULL* value, because the actual class of the table is often set as the default value and will be left as a *NULL* entry. Additionally we will filter out *boolean* and *date* type columns, because they are not adequate for subclass identification. Potentially *boolean* type columns are quite good to store subclasses in tables, for example an "isVolcano" column in the "mountain" table. But including *boolean* type columns in the identification process would introduce a subclass for almost all of them.

Furthermore to reduce the amount of unnecessary subclasses we define a percentaged threshold that each individual value in the examined column has to exceed to count as a subclass. For this threshold only non-*NULL* valued entries of the column are taken into account, because there will most likely be a huge amount of *NULL* values representing the default class of the table. We found that a threshold of 50% is enough to cut out most of the unnecessary potential subclasses.

One of the remaining columns that hold a subclass information is the "type" column of the "mountain" table. The unique values and their number of occurence in the column are as followed:

value	# of occurence
monolith	2
volcanic	36
volcano	65
NULL	148
Total without *NULL*	103

This means that "volcano" accounts for ~63,1% of all entries without *NULL* values and thus is eligible to be used as a subclass of *mountain*. To distinguish this subclass from other potential *volcano* classes, it is named *mountain_volcano.* This result is stored in the *AbstractSubClCols* metadata table:

AbstractSubClCols				
classname	property	localclassname	columnname	isclassvalue
mountain_volcano	type	mountain	type	volcano

The class *mountain_volcano* class has no hometable of its own and is thus regarded as an abstract class. This is why the $(subclass, superclass)$ pair is only added to the *AllCl* table:

AllCl	
subclass	superclass
mountain_volcano	mountain

An unexpected result was the "percentage" column of the "encompasses" table. The value "100" has a near 96% coverage of all table entries, which makes it a prime example for a subclass. Being a numeric value it would not bring much context if the subclass is simple named "encompasses_100". Therefore the column name is included in the subclass name resulting in *encompasses_percentage100*.

3.4.2 Deriving a class hierarchy from previous findings

As mentioned beforehand, deriving a class hierarchy just from the relational tables can be quite tricky. Comparing the RDF2SQL version of the relational MONDIAL database with the original relational database some assumptions can be stated.

First of all we can look at the difference in ranges of the ontology properties and the relational table columns. On the one hand the RDF ontology allows the user to define object-valued properties that range over multiple classes. On the other hand, the relational schema also allows the use of multiple FKs on the same column or combination of columns, but instead of achieving the union of all FKs reference targets, the intersection is applied as the constraint. This also applies if examining a single column that is part of multiple FKs but each FK has a different combination of columns. The "country" table is again taken as an example. The columns $(capital, code, province)$ form a FK to the "city" table and an additional FK with the columns $(capital, province)$ to the "province" table could be possible, too. This means that all values of the overlapping columns $(capital, province)$ have to be found in the "city" table and the "province" table. To achieve ranges like in the RDF/OWL ontology, a column with values from multiple

tables that do not overlap can not have a FK at all. The ranges have to be identified through the column entries as explained in Section 3.5.

If the analysis identifies such a column with entries referencing only a specific set of tables, we can assume that the column was deliberately constructed in such a way and named appropiately to describe its content. One example would be a "Water" column as it is used in the "locatedAt" table from the RDF2SQL version database. All column entries are references to either one of the tables "sea", "lake" or "river". It can be concluded from that that the classes *sea*, *lake* and *river* are subclasses of a class *Water*. *Water* has to be an abstract class, because no ClassTable is named like that, but for the *SubCl* table it is only important if the subclass is a concrete class. Thus the entries can be added in both tables:

AllCl	
subclass	superclass
river	Water
sea	Water
lake	Water

SubCl	
subclass	superclass
river	Water
sea	Water
lake	Water

Furthermore it would be benefical to find groups of classes like *GeographicalThings* (e.g. *Mountain, Desert*) or *PoliticalThings* (e.g. *Organization*, *Country*), but the tables in these groups do not have much distinctively in common to warrant a safe superclass definition. Even if the PK of one table is used in the PK of another table, we can not be sure if the class of one of those tables is the subclass of the other one. For example the "city" table which PK consists of the columns $(name, country, province)$ and contains two FKs $(country)$ to "country" and $(country, province)$ to "province". With only this evidence we can not assume that the three classes can be grouped together or that the class *city* is a sub- or superclass of *country* and *province*. On the other hand one grouping that could be possibly exploited is the special case of the CompositeNMTable. These NMTables are like multiple normal NMTables stacked together and have more than one FK on at least one side. We can conclude from that that the tables grouped on the same side have the same relationship with the table on the opposite side. The "located" table is again taken as an example with *city* on the first side and *river, lake* and *sea* on the second side. This would be another way to conclude that the classes *river*, *sea* and *lake* share a common superclass, but the "Water" name could not be concluded for it.

However going on from all the previous findings to derive a more complex class hierarchy is not possible. For example if two classes share the same subclasses and superclasses, we could not safely derive that these classes must be equivalent. This leaves us with a rather simple class hierarchy where the subclass relationships between most classes are not known.

3.5 Finding ranges for columns

For the previous steps, we assumed that all necessary FKs are given. As explained, the target of the FK is used as the range of the property that the columns in the FK are mapped to. Unlike properties defined in an ontology, these mapped properties are restricted to range over a single class, because multiple FKs on the same combination of columns in a relational database would only constraint them even further. However it is still possible that a column is filled in such a

way that it only contains entries referencing a specific set of multiple tables if no FK is defined.

To identify such ranges, every combination of columns in a table has to be examined including individual columns. The search can be limited by excluding every combination that already has a FK constraint or uses more columns than the biggest PK in the database. For each remaining combination, ClassTables with a PK that fits both in number of columns and datatype of columns are searched. Then, for each such table it is counted how many entries from the examined column combination and the PK overlap. If the examined column combination contains more than one column, each matching permutation with the PK columns has to be checked. For example if the examined column combination contains three *VARCHAR* type columns, six permutations have to be checked. In most cases only one permutation should yield overlapping data entries if at all, but to be sure the permutation with the most overlaps will be stored. After checking every permutation for all tables the number of overlapping entries is summed up and checked if all entries of the column combination were covered. This check can be screwed if multiple tables contain some identical PK entries. To minimize this problem no overlapping entries in ClassTableExtension tables will be searched. In the end if the overlap reaches 100% with a single target table, we can safely assume that the FK constraint was forgotten and add it. If multiple tables had overlaps they can at least be added as ranges to the column combination.

Therefore this mechanism can be used too if not all FK constraints are given beforehand so that the analysis can proceed as if they were. However if multiple tables have identical PK entries it is very likely that one table is the ClassTableExtension of the other one, but the RELMODELBUILDER can not solve which table is the correct one. This will result in both tables being defined as a ClassTable and mapped to an individual class. More importantly if only some PK entries overlap, no distinct FK can be defined to them from any other table. This is for example true for the "island" and "country" table, which share some PK entries like "Svalbard" or "Isle of Man". This means that for any other table that should have a FK to either of them, the FK cannot be defined which interferes with almost all of the previous explained steps and leads to the creation of a lot of false metadata entries.

When using the RELMODELBUILDER the "FindRanges" step can be disabled beforehand and should not be executed if not all FKs are given and overlapping PK entries exist.

Chapter 4

Implementation

In this chapter, the implementation of the RELMODELBUILDER tool is discussed. The program is written in JAVA using the JRE version 1.8 and does not utilise any other program besides the JDBC drivers and APACHE DBCP for accessing the databases. The first section covers how to use the RELMODELBUILDER class to create the inter-model mapping metadata for a specific database, while the remaining sections explain the individual conversion steps.

4.1 Using the RelModelBuilder class

The RELMODELBUILDER tool is a stand-alone program, which just uses a relational database as an input. The relational database is commited through the standard JDBC database URL and a $(User, Password)$ pair that at least gives read access to the specified database. For databases with multiple schemas the schema name has to be defined, too. Furthermore the name of the table owner is used to specify the set of tables that should be mapped to the ontology. So a simple constructor for the "mondial_rel" schema in a local "thedb" POSTGRES database can be called for example as such:

```
RelModelBuilder builder = new RelModelBuilder("jdbc:postgresql://localhost/thedb","
    scott","tiger","mondial_rel","postgres");
```

The RELMODELBUILDER supports relational databases in the ORACLE and POSTGRES DBMSs. Additionally there exist constructors to optionally transfer a boolean array as a parameter to define which of the conversion steps should be executed and which output is returned. These settings are coded by their position in the array. Positions 0 to 10 cover the output of the tool while positions 11 to 16 en- or disable optional conversion steps (see Appendix C). As default, all outputs and steps are enabled and should stay as such, but in case problems arise the user can tweak the conversion through these settings. One such case is the previously mentioned

problematic of missing FKs and overlapping PK entries. In this specific case it is recommended to disable the "FindFKs" step through setting position 16 in the boolean array as false.

After constructing an instance of the RELMODELBUILDER class the user can start the conversion process by calling the *convert()* function, which returns the *RelationalModel* object for the program interal inter-model mapping metadata representation:

```
RelationalModel rM = builder.convert();
```

If the conversion was successful the metadata tables with entries can be stored to the same or another database/schema by the "saveToDB" function:

```
builder.saveToDB("jdbc:postgresql://localhost/thedb","scott","tiger","metadata");
```

In this case, the metadata tables are not stored in the same schema, but to another "metadata" schema to keep them separate. This schema is automatically created if not already available. Thus the given user must have write access in the database and must be able to create schemas if this option is chosen.

4.2 Internal structures

This section explains all used classes for storing and managing the information necessary for the mapping.

4.2.1 RelationalModel

The *RelationalModel* from the DATABASECONVERTER tool is the main class for storing the metadata information of the mappings. It consists of objects storing the table entries for each metadata table discussed in Section 2.2.2 e.g. a *MappingDict* object to store the table entries of the MappingDictionary. Additionally it contains lists of *Table* objects that represent the relational tables and provide functions specific to the type of table represented. This thesis only utilises the *ClassTable* and *NMTable* subclasses to differentiate the tables. Each table has at least a name and a list of *Column* objects that in turn store information regarding each column of the table like its name and ranges.

ClassTable The *ClassTable* is the standard class to represent the relational tables. It covers ClassTables and ClassTableExtensions as well as ReifiedTables and SymmReifiedTables. A ClassTableExtension is identified by the HashSet<String> *extensionFor*, which stores the

names of the ClassTables it extends. If the table is a type of ReifiedTable, the sides the N:M relationship are stored in the *Column* lists *reifiedSide1* and *reifiedSide2*. It does not matter which side is assigned the first or second side.

NMTable The *NMTable* class stores information regarding NMTables and CompositeNMTables. Similar to ReifiedTables in a *ClassTable* object, the *NMTable* has to store the N:M relationship sides expressed by the table. However due to the special case of CompositeNMTables each side must be able to store multiple *Column* lists each representing one part of the NMSide.

4.2.2 TableSummary

The *TableSummary* object is a custom HashMap storing for each table name a list of *TableSummaryEntries*. Each *TableSummaryEntry* in turn stores all information that is available about a column in the corresponding table. The *TableSummary* is constructed despite the *Table* classes storing mostly the same information, because these classes were designed with the Ontology to relational Table mapping direction in mind and are missing crucial information for the conversion steps. However the *Table* classes are better representations for constructing the table entries of the inter-model mapping metadata after the analysis is done when the additional information is no longer needed. Thus the *TableSummary* focuses more on aiding the conversion process with easy access to almost all the information necessary for the following analysis steps and methods for recurring calculations. Most of these methods involve managing and analysing the structure of FKs in a table, because storing each column individually makes dealing with potential multi-column FKs more difficult . The most important methods are briefly explained in Table 4.1.

Method	Feature
findBothSidesOfNM	This function tries to find a two-way separation of the input columns of the given table, such that each side could be a side of a N:M relationship.
findReifiedSidesOfNM	This function tries to find a two-way separation of the functional columns in the table, so that they make up both sides of a N:M relationship.
separateFKs	Splits the FKs in the specified table and groups them based on the referenced table.

Table 4.1: Important methods of the *TableSummary* class

The method *findBothSidesOfNM* is used to identify NMTables while *findReifiedSidesOfNM* is used to identify ReifiedTables. Both methods make sure that the columns used for either side of the N:M-relationship have differences. Optimally we would want a clean two-way separation of the columns to the two sides of the relationship. So that columns (a, b, c) are the domain-side and columns (d, e) represent the range-side. However due to the fact that the same column can be used in a different FK, it can happen for multi-column FKs that columns overlap for both sides. Thus, it has to checked if the different FKs have at least one column not in common. Additionally for NMTables all columns of the table have to be involved in either of the sides. The *separateFKs* method is often used to get a well-structured overview over all FKs of a table. If a column is part

of multiple FKs it is represented with each of them. This method is especially helpful if the table has multiple FKs referencing the same table like recursive NMTables.

TableSummaryEntry A *TableSummaryEntry* is a comprehensive summary of every basic information available for a column in a table. Unlike the *Column* class a *TableSummaryEntry* most notably additionally stores the referenced $(Table, Column)$ pairs if involved in one or more FKs through a list of *TableCoumnPair* auxiliary objects. Furthermore it stores the abstract subclasses explained in Section 3.4.1 identified through this column.

4.3 Conversion steps

The conversion steps explained in the following are the main operations of the RELMODELBUILDER tool. They are executed in the order of explanation.

4.3.1 Loading table metadata from DBMS

As mentioned before, the RELMODELBUILDER tool supports ORACLE as well as POSTGRES DBMSs, but this section will only cover the access to a Postgres database. The process for ORACLE DBMS is identical with the exception that ORACLE uses a different structure to store meta information about the user tables, which results in slightly different queries.

At the end of the process all the necessary meta information about the tables is collected in a *TableSummary* object. Only operations on the table entries have to be done by connecting to the database, reducing the amount of time intensive connection initialisations.

The first step to gather this information is to access the DBMS internal tables storing the meta information about the user tables. This is split into three separate queries, each accessing different kinds of information and then combining those results into the creation of the *TableSummary*. Additional information is then added to it throughout the analysis steps.

The first query accesses all column names, their corresponding table name and their simple datatype. These datatypes include names of user defined complex datatypes like “geocoord”, but will not be split into their components, e.g. “latitude” and “longitude” both with datatype *numeric*.

```
SELECT c.relname as table_name, a.attname as column_name, t.typname as data_type
FROM pg_class c, pg_attribute a, pg_namespace ns, pg_type t, pg_authid auth
WHERE c.relkind = 'r' AND c.relname !~ '^(pg_|sql_)'
  AND c.relnamespace = ns.oid AND ns.nspname = '" + accessInfo.getSchemaname() + "'
  AND c.oid = a.attrelid AND t.oid = a.atttypid
  AND c.relowner = auth.oid
  AND auth.rolname = '" + owner + "'
ORDER BY 1, 2;
```

The important tables in this query are:

pg_class Covers the basic information regarding everything that is similar to a table and acts as the primary conjunction table for all other metadata tables

pq_attribute Stores all information regarding a column (attribute) of a table

pg_type Stores detailed information about the datatype of a column

All entries in these tables are identified by their internal object ID (oid), which are used to join the tables. To filter out everything else besides actual relational tables the "pq_class.relkind" attribute has to be set to 'r', which stands for "relation". However this still includes a whole lot of system internal tables, which have to be further filtered by excluding everything starting with the name "pg_" or "sql_". Furthermore the query sets the table owner through the "pg_authid" table filtering out the rest of the internal tables. If the database contains multiple schemata the schema name can be set through the "pg_namespace" table.

The second query retrieves all PK columns for each table.

```
SELECT c.relname as table_name, a.attname as column_name
FROM pg_class c, pg_attribute a, pg_index i, pg_namespace ns
WHERE c.relkind = 'r' AND c.relname !~ '^(pg_|sql_)'
  AND c.relnamespace = ns.oid AND ns.nspname = '" + accessInfo.getSchemaname() + "'
  AND c.oid = a.attrelid AND a.attrelid = i.indrelid
  AND a.attnum = ANY(i.indkey) AND i.indisprimary IS TRUE
ORDER BY 1, 2;
```

The query utilises again the tables "pg_class", "pg_attribute" and "pg_namespace" to target all *(table, column)* pairs in a specified schema. In addition to that the "pg_index" is used to select only columns involved in a PK. This table stores all indices defined over all columns in the tables. Fortunately defining a PK constraint over columns also automatically creates an index over those columns for faster lookup operations. These special PK indices are marked in the "pg_index" table by the "indisprimary" attribute which is set to true. The columns in the index are not stored by their name, but by their position in the table, which means that the column number "pg_attribute.attnum" has to be any of the numbers in the "pg_index.indkey" array.

In the end the third query retrieves all FK columns of each table and their respectiv column target in the referenced table.

```
SELECT cl2.relname AS ForeignTable, att2.attname AS ForeignColumn, cl.relname AS
    PrimaryTable, att.attname AS PrimaryColumn
FROM (SELECT unnest(con1.conkey) AS parent, unnest(con1.confkey) AS child, con1.
    confrelid, con1.conrelid
          FROM pg_class cl
                JOIN pg_namespace ns ON cl.relnamespace = ns.oid
                JOIN pg_constraint con1 ON con1.conrelid = cl.oid
          WHERE ns.nspname = '" + accessInfo.getSchemaname() + "'
                AND con1.contype = 'f') con
```

```
JOIN pg_attribute att ON att.attrelid = con.confrelid AND att.attnum = con.child
JOIN pg_class cl ON cl.oid = con.confrelid
JOIN pg_attribute att2 ON att2.attrelid = con.conrelid AND att2.attnum = con.parent
JOIN pg_class cl2 ON att2.attrelid = cl2.oid;
```

The main part of this query is the nested SELECT-query labeled "con". It utilises the "pg_constraint" table, which stores information about all defined constraints. Theoretically the PK constraints could have been accessed in this table too, but using the "pg_index" table is more straightforward. The table where the constraint is defined on is identified with the "conrelid" attribute and the affected column with the "conkey" attribute. On the other side of the constraint the referenced table and column are identified by the "confrelid" and "confkey" attributes respectively.

4.3.2 Initialisation of Table objects

As explained in Section 4.2, constructing a *ClassTable* or an *NMTable* object for each relational table and filling them with the correct information is the objective of the conversion steps before converting them into the metadata table entries as the final goal. After loading the basic information about the relational tables from the database we decided it would be the best approach to first construct initial objects for each table which will be further fleshed if new information becomes available during the following steps. This procedure is supported among other things by the fact that most of the tables in the relational database will end up being *ClassTable*s, because only tables representing a pure N:M relationship without the need of reification are stored as *NMTable*s. Thus for the initial differentiation every table will be stored as a standard *ClassTable* except those that are identified as pure N:M tables by consisting of two distinct FKs, which are disjoint but together make up the whole table. Additionally, if a PK is present it must cover all columns.

This means that there will be errors in this initial guess which have to be resolved during the conversion steps. These include for example identifying and marking all the special cases of ClassTables, for instance ReifiedTables. Furthermore with the advanced possibilities of ranges compared to only FKs it is possible that some N:M tables are identified later on and the *ClassTable* object has to be converted to a *NMTable*, which will be discussed in Section 4.3.4.

4.3.3 Handling of object-valued properties

When mapping columns to properties, some of these properties can be object-valued. The *handleObjectValued*() function covers all sub-functions that identify the ranges for these object-valued properties.

First of all, all properties mapped from the columns involved in a FK are guaranteed to range over the target table of the FK. To do this, the RELMODELBUILDER iterates over all *TableSummaryEntries* and adds the table name of the *FKTarget* to the column ranges. Of course a property ranges over a class and not a relational table, but if the FK constraints are correctly used as

assumed, they should reference only to ClassTables which are directly mapped to classes having the same name.

To identify further object-valued properties from columns that are not involved in FKs, it is necessary to check each combination of columns in a table whether their entries reference to other ClassTable PKs. To check each combination, the *generateSubsets*(*int* minimum, *int* maximum, *LinkedList<TableSummaryEntry>* input) method is used to generate a list of column subsets from the input whose sizes range from the specified minimum to maximum number of entries. The maximum number of entries is set to match the longest PK available in the database. The call $generateSubset(1, 3, name, country, province)$ for example would then return the subsets:

$$generateSubset(1, 3, name, country, province) =$$
$$\left\{ \begin{array}{ccc} \{name\} & \{country\} & \{province\} \\ \{name, country\} & \{name, province\} & \{country, province\} \\ & \{name, country, province\} & \end{array} \right\}$$

The position of the items in each subset does not matter, because the datatypes of the columns are also important. The next step tries to find potential ranges for each column combination by joining them to the PK columns of matching ClassTables. However it is for example not possible to join a *numeric* column with a *Varchar* column. Thus the *findRanges* method has to first off check for ClassTables with equal amount of columns for each datatype in the PK as in the examined column combination. After that it has to construct all possible column matchings that the datatypes allow. For instance a column combination with two *Varchar* columns and one *numeric* column would have two possible matchings with a suitable ClassTable, because the *numeric* columns have to be matched and the two *Varchar* columns allow for two variations. Therefore with increasing amounts of columns especially if they have the same datatype the number of column combinations and column matchings that have to be checked increase exponentially. Fortunately using more than three or four columns for a table's PK is not common.

The next step is to use the column matchings and to query the database for overlapping column entries. In most cases only one of the column matchings will return overlapping entries if any at all. If there are multiple possible matchings for a single table, the matching with the highest overlap percentage is choosen and the column matching with the percentage stored in a *RangeInformation* object for easier recap of the analysis results. In the end one *RangeInformation* object will be available for each suitable table and evaluated together. If only one table returned a column entry overlap and the overlap was 100%, it is basically a missing FK and will be marked as such by adding the column matchings to the foreign key targets of the respective *TableSummaryEntry*s. On the other hand, if multiple tables returned matching column entries, these tables will be added to the *ranges*.

4.3.4 Identification of NMTables

The initial *NMTables* were only constructed for pure N:M tables with the clear pattern of two disjoint FKs and no additional columns. After identifying the *ranges* of columns and missing FKs however it is possible to identify further *NMTables* that were previously missed. Essential for this is the *findBothSidesOfNM* method of the *TableSummary*. It analyses the column data and tries to find a two-way separation of the columns such that:

- Each side represents a range or FK
- The columns in one side are disjunct to the columns in the other side
- The columns of both sides form together the whole table
- Optional: The column entries for each side stand in a N:M relationship to the other sides entries

The easiest case for this is if only two columns are present in the table where each column has to have an individual FK or range. This is the case for standard NMTables like "mountainOnIsland" as shown in Figure 4.1 and databases that follow the URI pattern of an RDF ontology. After all the purpose of this step is not only to identify new NMTables, but also to add the NM-sides to all new and old *NMTable* objects.

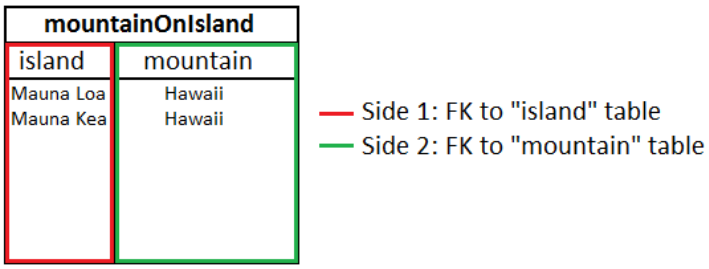


Figure 4.1: NM-sides of the "mountainOnIsland" table from the relational MONDIAL database

On the other hand if the table has more than two columns there has to be at least one multi-column FK. For these cases, all the FKs of the table are first split by the *separateFKs* method and then each combination of two FKs is tested if they fulfill the mentioned requirements. This applies for example for the "locatedOn" table as shown in Figure 4.2.

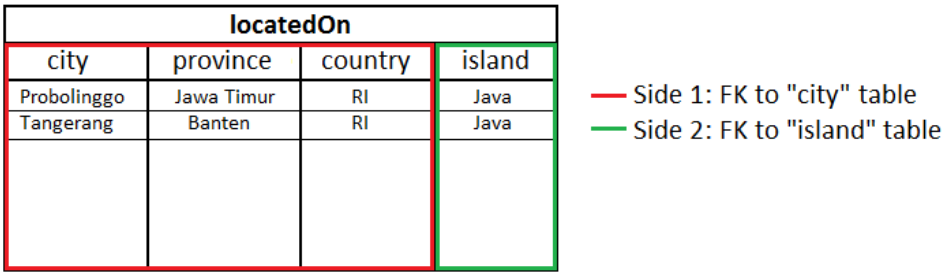


Figure 4.2: NM-sides of the "locatedOn" table from the relational MONDIAL database

Even if no such combination could be found it could still be possible that the table is an NMTable. This is the case if one side is neither a FK nor has a range, thus is a literal-valued column. In this setup the remaining columns have to be calculated for each FK and then the combination has to be checked.

For tables with only two distinct FKs that were already stored as an *NMTable* object, it is pretty much garantueed that they represent an N:M relationship and thus it is not necessary to further check the table entries. This helps to soften the requirements, because not all NMTables actually have N:M column entries for both directions. However when checking previously declared *ClassTable*s if they actually are NMTables, it is best to be more careful by adding this constraint and thus avoiding false identifications just because a combination of FKs or ranges fit the pattern.

If a ClassTable is found to be actually an NMTable, the previous *ClassTable* object has to be exchanged for a new *NMTable* object and all necessary information about the columns has to be copied. This means that if multiple ranges or FKs exist, only those are kept that are used to define the two NM-sides. The sole purpose of an NMTable is the representation of the N:M relationship between the two sides, so additional references to other tables are not wanted. This happens for example for the multi-column FK on the first side of the "locatedOn" table shown in Figure 4.2. All three columns together are part of the FK to the table "city", however after identifying additional references in Section 4.3.3 there will be two new FKs in the table. One FK with columns $(province, country)$ references to the "province" table and the other FK consisting just of the $(country)$ column references the "country" table. Those FKs could have been helpful for the analysis, but have to be deleted now after they were deemed redundant. Furthermore all references to the new NMTable from other ClassTables have to be deleted, but in the standard case there should not have been one to begin with. To finish the transformation, the tablename has to be deleted from all previously found class hierarchy entries, because it will no longer be mapped to an ontology class.

After checking the standard NMTable pattern for a table and no distinct NM-sides could be found, it could still be possible that the table is a CompositeNMTable like the "located" table shown in Figure 4.3.

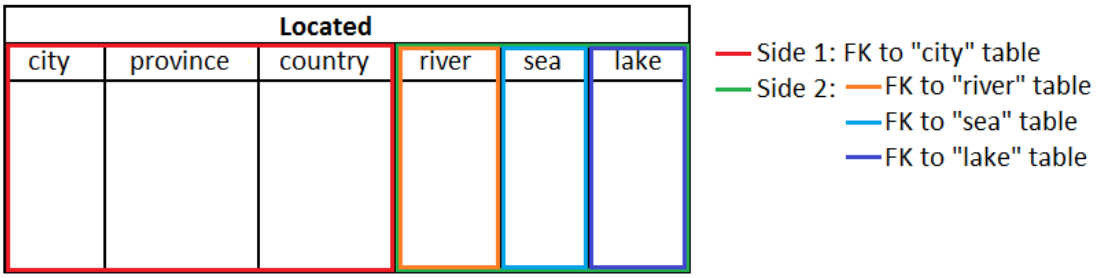


Figure 4.3: NM-sides of the "located" table from the relational MONDIAL database

Because this kind of table is like a fusion of multiple individual tablies, it is possible to identify them by first identifying all subparts. In this case this would be the three N:M relationships $(city, river)$, $(city, sea)$ and $(city, lake)$. The process starts with separating the FKs and range columns again, but then generating all possible subsets of size 2 from them similar to the procedure in Section 4.3.3. Each combination is then tested individually on the requirements of a NMTable. It should be noted that CompositeNMTables tend to have a *dominant* side that forms the connection point between the individual NMTable subparts, which is in this case the

"city" side. Because of that and the fact that not all entries on the dominant side have an entry for each of their counterparts it is highly likely that the weaker side has lots of *NULL* values in their columns. In the highlighted example almost all of the cities are located on just one of the three water types and in rare cases on two, but never on all three. This means that the false combinations $(river, sea)$, $(river, lake)$ and $(sea, lake)$ can be easily excluded by checking if there exist any $(NULL, NULL)$ entries, which should never be the case in an NMTable. In a standard NMTable $(Entry, NULL)$ entries would also be incorrect, but those have to be tolerated for a CompositeNMTable. All other combinations that pass the N:M relationship requirements are added to a list and in the end it is checked if all columns of the table were used. If this is the case, the *ClassTable* object is again transformed into an *NMTable* object, all unnecessary references are deleted and the NM-sides for each NMTable subpart are added. The overall dominant NM-side can later on be determined again with the *findDominantMultiNMSide* method if necessary.

4.3.5 Identification of ClassTableExtensions

After identifying all NMTables, only the special cases of ClassTables are left starting with the ClassTableExtensions. We defined ClassTableExtensions in Section 3.2.1.2 to be ClassTables with a 1:1 connection to another ClassTable and there exist no references from other tables to it. For this definition to work, all necessary FKs in the database had to be specified before the *handle object-valued properties* step, because newly detected references would range over both the ClassTableExtension and the extended ClassTable instead of just the main extended ClassTable. This would also mean in practice that ClassTableExtensions and their extended ClassTables would reference each other making the destinction between the two impossible.

However if the references between tables are correct, the identification of ClassTableExtensions is quite simple. First off check for each remaining ClassTable if there exists a reference from any other table to it. If this is not the case then verify if the PK columns of the examined table reference only one table. This is done by looking up the previously stored FK constraints of the columns, because these guarantee a 100% overlap of the column entries in the potential ClassTableExtension to the extended ClassTable. The examined table is only a ClassTableExtension if there are no additional column entries besides the references to the extended table, thus ranges are not allowed. The extended ClassTable is stored in the *extensionFor* variable of the ClassTableExtension, which automatically serves as an indicator, too.

4.3.6 Identification of ReifiedTables

The last special case of ClassTables that have to be identified are ReifiedTables. As explained in Section 3.1.3, ReifiedTables are similar to NMTables with the exception that they store additional columns that are not part of either of the two NM-sides. Thus identifying them is also quite similar to the process highlighted in Section 4.3.4. The method *findReifiedSidesOfNM* of the *TableSummary* class provides the same functionality as the *findBothSidesOfNM* method for identifying NMTables, but tailored to the special requirements of ReifiedTables. Instead of using all columns in the table the function takes only the columns in the PK if available or all columns with ranges or part of FKs if not. The rest of the process is analogous to the two-way separation of identifying NMTables.

However there are some differences after the identification of the NM-sides of the ReifiedTable. Although the *ClassTable* is not transformed into an *NMTable* object, the identified NM-sides still has to be stored in the *reifiedSide1* and *reifiedSide2* lists. The order of the NM-sides does not matter. Multiple parts of an NM-side as with CompositeNMTables are not allowed for ReifiedTables. But the surplus references still has to be pruned though only for the columns involved in the NM-sides. The remaining columns and their information stay untouched. Furthermore unlike standard NMTables it is more important if the ReifiedTable represents a symmetric relationship or not. A SymmReifiedTable like the "borders" table shown in Figure 4.4 can be easily distinguished if both NM-sides reference to the same target table and the table entries for both sides are symmetric.

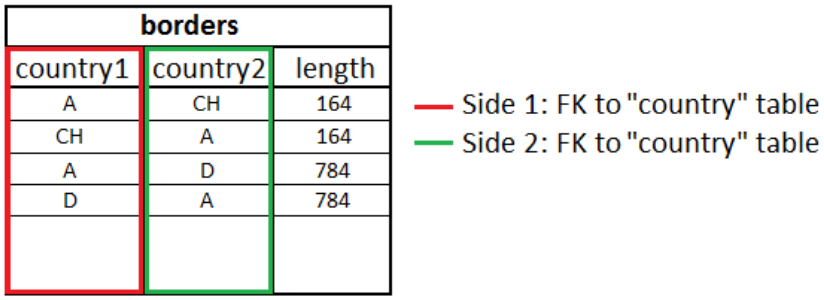


Figure 4.4: NM-sides of the "borders" table from the relational MONDIAL database

The *isSymmReified* variable is set to true to mark such a table. However finding the name of the property that the relationship will be mapped to is not as clear. Due to the fact that the property is symmetric and thus is the inverse of itself only one name is necessary for both NM-sides. We assume that the columns for the NM-sides are named quite similar as is the case with the "country1" and "country2" columns in the "borders" exmaple. The common property name is then the longest possible prefix that matches the columns on both sides. For instance this would be "country" for these columns. This name is then stored in the *symmReifiedColName* variable for later use.

4.3.7 Identification of N:1 tables

The last table type that needs to be identified are so called N1Tables. In the relational database only the "cityothername","countryothername" and "provinceothername" tables are instances of such tables. The "cityothername" table is illustrated in Figure 4.5.

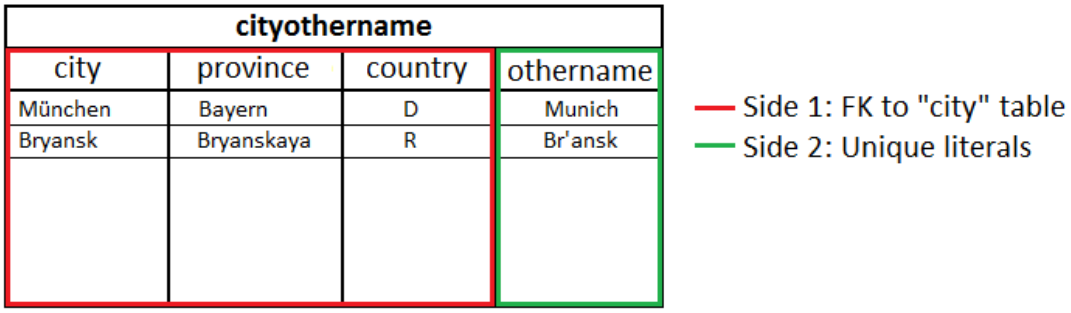


Figure 4.5: NM-sides of the "cityothername" table from the relational MONDIAL database

Their origin is the *othername* property from the RDF ontology of MONDIAL. In the forward mapping from ontology to relational tables this property is represented as an NMTable "othername", which stores for URIs of several different classes their real world name. This means that only one NM-side is a reference while the other one is just a literal. In the relational MONDIAL database this information is split into the three previously mentioned tables because there is no direct superclass of *city*, *country* and *province*. Those tables do not fullfil all requirements of a true NMTable most notably that they only represent an N:1 relationship. Such information would normally be stored in the home table representing the functional or unique side of the (1:N) - relationship. For example the *hasHeadquarter* property of the *organization* class. The headquarter of an organization can only be located in one city, but a city can host headquarters of multiple organizations. Hence this information is stored on the *organization* side. However the functional side of the othername case is just a literal value and thus there exists no table that could store this information from the functional side. Therefore, a new table is required storing the information from the non-functional side.

The occurences of these circumstances are so rare that they do not warrant the definition of a new kind of table, but luckily they can be treated as normal NMTables with just one NM-side. This is also the reason why they are covered as the last special case, because the previous table types are more significant. The identification process is then similar to standard NMTables. Iterate over all *ClassTable*s that were not identified as one of the previously explained special cases and check all columns of the PK again with the *findBothSidesOfNM* method. The method is able to identify a literal valued column as one of the NM-sides and will return the sides accordingly. If the sides are returned successfully the function then checks if any references targeting this table exist as with standard NMTables. After that it calculates the references of each side where one side has to target at least one table while the other side has to target none indicating the literal-valued column. If this is the case then an N1Table was identified and the *ClassTable* object is transformed into an *NMTable* object including the reference pruning if necessary.

4.3.8 Derivation of Subclasses

The subclass and superclass relationships are stored in two separate *ClassHierarchy* objects in the *RelationalModel* object. One instance called *completeclasshierarchy* (later on the *allcl* table) stores all available information while another instance *concreteclasshierarchy* (later on the *subcl* table)

stores only entries where the subclass is a concrete class. As mentioned in Section 3.4 those class hierarchies have to have at least one entry for each concrete class namely all remaining ClassTables. Thus the two lists are initialised with one entry per ClassTable as the subclass and the "rdf2sqlTop" placeholder token as the superclass.

Before the initialisation, the *handle object-valued properties* step from Section 4.3.3 could have already added some entries depending on the findings. If ranges were identified for a column or column combination, the ranges form a group of subclasses with the column as the superclass. The name of such columns is usually a fitting description for the superclass of the classes grouped in its ranges. This means for instance for a column "Water" with ranges *sea*, *lake* and *river* that the class hierarchy entries $(sea, Water)$, $(lake, Water)$ and $(river, Water)$ are produced. Because *Water* is not the name of a ClassTable it is defined as an abstract class, but the entries are nevertheless added in both *completeclasshierarchy* and *concreteclasshierarchy* because only the subclasses have to be concrete. To prevent frequent entries like $(country, country)$ all entries are filtered out where the subclass equals the superclass.

The next step is to identify abstract subclasses inside of ClassTables as explained in Section 3.4.1. For this the *RelModelBuilder* has to iterate over all literal-valued columns of ClassTables and examine each one independently. Object-valued columns were already covered in the previous step and are thus excluded just as are columns with datatype *boolean* or *Date*. For each remaining column a HashMap is created that stores all unique values with their number of occurences and the number of total entries without *NULL* values is calculated. As explained in Section 3.4.1 the coverage of a unique value has to exceed a lower boundary to be deemed a good representative of an abstract subclass. We found a boundary of 50% to be a good balance between too many and no subclasses. This boundary also means that there can be only a maximum of two subclasses per column if the entries are perfectly split. Setting the boundary any lower opens up the possibility of a lot more subclasses which is not necessarily a bad thing if more differentiation is wanted, but setting it higher emphasises the meaningfulness of the newly defined subclasses. The name of the new subclass is derived from the value in the column that is used to define it. The values of *VarChar* type columns can be used directly, but numeric values are not suitable as a classname. For such cases the column name is added to the value. Furthermore regardless of type the table name is added as an affix to the classname to underline the affiliation of the abstract subclass to the class of the table. If the examined table is a ClassTableExtension the name of the extended table is of course taken instead. These class hierarchy entries are only added to the *completeclasshierarchy*, because the subclass is always regarded as an abstract class.

The final step would be to analyse the *completeclasshierarchy* and *concreteclasshierarchy* entries and derive relationships that are implied by them. However as explained in Section 3.4.2 we have sadly found no way to derive certain facts from this limited information.

Chapter 5

Conclusion

Making storing and querying relational and RDF data interchangeable is a good step for using the favorable characteristics of both at the same time. The *RelModelBuilder* fills the gap of automatically creating an RDF ontology from any relational database including the mappings stored in the inter-model mapping metadata tables. This functionality will for instance aid the use of our previous tools like the *QueryConverter* to state *SPARQL* queries based on the created ontology to the relational database. Furthermore new tools will be able to build on this possibility like the mentioned *SchemaMatcher* that can be used to create mappings between inter-model mapping metadata of multiple relational databases with similar schemas to access them all at once. This interchangeability can only be achieved because of separation of the mappings from the tools and storing it independently in the metadata tables.

The *RelModelBuilder* tries to extract the most accurate ontology from the relational database, however there are still some flaws in the resulting ontology. First of all some of these shortcomings are unavoidable due to the automatic nature of the approach. For example unlike semi-automatic approaches which are partly guided by the user, the program has to choose property names for itself. Taking the column names for their representative property turned out to be a suitable choice in most cases, but for instance property names for multi-column FKs are not comprehensive for the user. Such properties should be renamed before extensive usage. We plan to improve the automatic naming schema and develop a user interface that provides an easy access point for renaming properties in all inter-model mapping metadata tables. Furthermore it would be beneficial to have access to a more extensive class hierarchy and more differentiation in abstract superclasses. However such information is just not available from a plain relational database. Even if the database was constructed on the basis of an ER model with such features, they are not clearly represented in the database alone.

Overall the *RelModelBuilder* procudes a good basis to access and work with the database on an ontology level. Unfortunatly, we cannot simply compare our results with those produced from the tested tools in the RODI benchmark, because their internal datastructures for the OBDA mapping are not known to us. The quality of the OBDA mappings in the benchmark were only measured indirectly through the results of ontology alignment and query conversion. However the goal for this work was to generate the OBDA mappings in our internal model for the SCHEMAMATCHER tool. We can then combine the OBDA mapping generation of the *RelModelBuilder*, the ontology alignment of the *SchemaMatcher* and the SPARQL-to-SQL query

transformation of the *QueryConverter* to test our tools under the same conditions as in the RODI benchmark.

5.1 Future works

The OBDA mapping the *RelModelBuilder* generates for a relational database can potentially be utilised for more then just the SPARQL-to-SQL query rewriting of the *QueryConverter*. They can also be used to change the database structure with so-called atomic schema operators. These include for example changing functional properties into non-functional ones and automatically converting the associated table structures and metadata table entries. After defining such operators, it would be possible to use the ontology alignment from two similar databases produces by the *SchemaMatcher* as a step by step guide to converge the database schemas.

Bibliography

[ADMR05] D. Aumueller, H.-H. Do, S. Massmann, and E. Rahm. Schema and Ontology Matching with COMA++. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pp. 906–908, New York, NY, USA, 2005. ACM. 1.2

[Age] C. I. Agency. The World Factbook. Website. Available online at `https://www.cia.gov/library/publications/the-world-factbook/`; visited on October 11th 2016. 1.5

[BCH+14] T. Bagosi, D. Calvanese, J. Hardi, S. Komla-Ebri, D. Lanti, M. Rezk, M. Rodríguez-Muro, M. Slusnys, and G. Xiao. *The Ontop Framework for Ontology Based Data Access*, pp. 67–77. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. 1.2

[CB74] D. D. Chamberlin and R. F. Boyce. SEQUEL: A Structured English Query Language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, pp. 249–264, New York, NY, USA, 1974. ACM. 2.1.2

[Che76] P. P.-S. Chen. The Entity-relationship Model&Mdash;Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976. 2.1.1

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, June 1970. 2.1.2

[dMPC15] L. F. de Medeiros, F. Priyatna, and O. Corcho. *MIRROR: Automatic R2RML Mapping Generation from Relational Databases*, pp. 326–343. Springer International Publishing, Cham, 2015. 1.2

[GEO] GeoHive. Website. Available online at `http://www.geohive.com/`; visited on October 11th 2016. 1.5

[Groa] R. W. Group. RDF Homepage. Website. Available online at `http://www.w3.org/RDF`; visited on October 15th 2016. 2.1.3

[Grob] T. P. G. D. Group. PostgreSQL Download. Website. Available online at `https://www.postgresql.org/download/`; visited on October 11st 2016. 1.4

[Groc] T. P. G. D. Group. PostgreSQL Homepage. Website. Available online at `https://www.postgresql.org/`; visited on October 11st 2016. 1.4

[HM12] T. Hornung and W. May. Efficient, Schema-Aware Storage of RDF. Technical report, Universität Freiburg, Institut für Informatik, Universität Göttingen, Institut für Informatik, 2012. 2.2.1, 3.1

[JRKZ+15] E. Jiménez-Ruiz, E. Kharlamov, D. Zheleznyakov, I. Horrocks, C. Pinkel, M. G. Skjæveland, E. Thorstensen, and J. Mora. *BootOX: Practical Mapping of RDBs to OWL 2*, pp. 113–132. Springer International Publishing, Cham, 2015. 1.2

[KSA+12] C. A. Knoblock, P. Szekely, J. L. Ambite, A. Goel, S. Gupta, K. Lerman, M. Muslea, M. Taheriyan, and P. Mallick. *Semi-automatically Mapping Structured Sources into the Semantic Web*, pp. 375–390. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. 1.2

[May] W. May. The Mondial Database. Website. Available online at `https://www.dbis.informatik.uni-goettingen.de/Mondial/`; visited on October 11th 2016. 1.5

[May99] W. May. Information Extraction and Integration with FLORID: The MONDIAL Case Study. Technical Report 131, Universität Freiburg, Institut für Informatik, 1999. Available from `http://dbis.informatik.uni-goettingen.de/Mondial`. 1.5

[Oraa] Oracle. Java Homepage. Website. Available online at `http://www.oracle.com/technetwork/java/index.html`; visited on October 11st 2016. 1.4

[Orab] Oracle. SQL Developer Homepage. Website. Available online at `http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html`; visited on February 1st 2014. 2.1.2

[PBJR+15] C. Pinkel, C. Binnig, E. Jiménez-Ruiz, W. May, D. Ritze, M. G. Skjæveland, A. Solimando, and E. Kharlamov. *RODI: A Benchmark for Automatic Mapping Generation in Relational-to-Ontology Data Integration*, pp. 21–37. Springer International Publishing, Cham, 2015. 1.2

[PBKH13] C. Pinkel, C. Binnig, E. Kharlamov, and P. Haase. IncMap: Pay As You Go Matching of Relational Schemata to OWL Ontologies. In *Proceedings of the 8th International Conference on Ontology Matching - Volume 1111*, OM'13, pp. 37–48, Aachen, Germany, Germany, 2013. CEUR-WS.org. 1.2

[RS13] L. Runge and S. Schrage. Forschungspraktikumsbericht RDF-to-SQL-Konverter. Technical report, Universität Göttingen, Institut für Informatik, 2013. 1.3, 2.2.1

[RS14] L. Runge and S. Schrage. Auswertung von SPARQL-Anfragen mit relationaler Speicherung. Technical report, Georg-August-Universität Göttingen, Zentrum für Informatik, 2014. Available from `https://webhelper.informatik.uni-goettingen.de/editor/media/theses/2014/ZAI-BSC-2014-09-runge.pdf`; visited on October 13th 2016. 2.1.3, 2.2.1

[Sch16] S. Schrage. Transformation-based Ontology Mapping. Technical report, Georg-August-Universität Göttingen, Zentrum für Informatik, 2016. Work in Progress. Submission date November 30th, 2016. 1.3, 2.2.3

[W3Ca] W3C. A Direct Mapping of Relational Data to RDF W3C Recommendation. Website. Available online at `https://www.w3.org/TR/rdb-direct-mapping/`; visited on November 14th 2016. 1.2

[W3Cb] W3C. W3C Notation3. Website. Available online at https://www.w3.org/TeamSubmission/n3/; visited on October 12th 2016. 2.1.3

[W3Cc] W3C. W3C OWL 2 Web Ontology Language Recommandation. Website. Available online at https://www.w3.org/TR/2009/REC-owl2-syntax-20091027/; visited on October 15th 2016. 2.1.4

[W3Cd] W3C. W3C OWL Web Ontology Language Current Status. Website. Available online at https://www.w3.org/standards/techs/owl; visited on October 15th 2016. 2.1.4

[W3Ce] W3C. W3C RDF 1.1 Recommendation. Website. Available online at https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/#section-Introduction; visited on October 12th 2016. 2.1.3

[W3Cf] W3C. W3C RDF Schema 1.1 Recommendation. Website. Available online at https://www.w3.org/TR/2014/REC-rdf-schema-20140225/; visited on October 12th 2016. 2.1.4

[W3Cg] W3C. W3C RDFS Publication History. Website. Available online at https://www.w3.org/standards/history/rdf-schema; visited on October 15th 2016. 2.1.4

[W3Ch] W3C. W3C Recommendation. Website. Available online at http://www.w3.org/TR/rdf-sparql-query; visited on October 13nd 2016. 2.1.3

[W3Ci] W3C. W3C XML. Website. Available online at https://www.w3.org/XML/; visited on October 12th 2016. 2.1.3

Appendix A

Definition of abstract/concrete classes: mondial-er.n3

```
#@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:Entity er:isa er:Abstract.
: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.

####### Abstract Classes

### choose to have each of them  Concrete or Abstract:
#:MondialThing er:isa er:Concrete.
#:ReifiedThing er:isa er:Concrete.
#:Area er:isa er:Concrete.
#:Location er:isa er:Concrete.
#:Line er:isa er:Concrete.
```

```
#:Place er:isa er:Concrete.
#:PoliticalThing er:isa er:Concrete.
#:GeographicalThing er:isa er:Concrete.
#:AdministrativeArea er:isa er:Concrete.
#:PoliticalBody er:isa er:Concrete.
#:GeographicalNonPoliticalThing er:isa er:Concrete.
#:AnthropoGeographicalThing er:isa er:Concrete.
#:AnthropoGeographicalRelationship er:isa er:Concrete.
:MondialThing er:isa er:Abstract.
:ReifiedThing er:isa er:Abstract.
:Area er:isa er:Abstract.
:Location er:isa er:Abstract.
:Line er:isa er:Abstract.
:Place er:isa er:Abstract.
:PoliticalThing er:isa er:Abstract.
:GeographicalThing er:isa er:Abstract.
:AdministrativeArea er:isa er:Abstract.
:PoliticalBody er:isa er:Abstract.
:GeographicalNonPoliticalThing er:isa er:Abstract.
:AnthropoGeographicalThing er:isa er:Abstract.
:AnthropoGeographicalRelationship er:isa er:Abstract.

####### 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:Abstract.    ## <<<<<<<<<<<<<<<<<<<
#:Sea er:isa er:Abstract.
#:River er:isa er:Abstract.
#:Water er:isa er:Concrete.   ## <<<<<<<<<<<<<<<<<<<
:Lake er:isa er:Concrete.
:Sea er:isa er:Concrete.
:River er:isa er:Concrete.
:Water er:isa er:Abstract.   ## <<<<<<<<<<<<<<<<<<<
: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.
```

```
:Islands er:isa er:Concrete.
:Mountains er:isa er:Concrete.

er:SymmetricReifiedRelationship rdfs:subClassOf er:ReifiedRelationship.
:Border er:isa er:SymmetricReifiedRelationship;
    er:reifies :neighbor.

:Volcano er:isa er:Abstract.

:Encompassed er:isa er:ReifiedRelationship;
    er:reifies :encompassed.
:Membership er:isa er:ReifiedRelationship;
    er:reifies :isMember.
:SpokenBy er:isa er:ReifiedRelationship;
    er:reifies :speakLanguage.
:BelievedBy er:isa er:ReifiedRelationship;
    er:reifies :believeInReligion.
:EthnicProportion er:isa er:ReifiedRelationship;
    er:reifies :belongToEthnicGroup.
```

Appendix B

Definition of classes and properties: mondial-meta.n3

```
#@prefix : <http://www.semwebtech.org/mondial/10/meta#>.
@prefix : <f://m#>.
@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#>.
# Design:
# class hierarchy described by disjointUnionOf as far as possible.
# some declarations are intensionally redundant

# ... and now to the Mondial ontology

:MondialThing a owl:Class;
    owl:disjointUnionOf   ( :PoliticalOrGeographicalThing
                            :AnthropoGeographicalThing
                            :ReifiedThing ).

:PoliticalOrGeographicalThing a owl:Class;
    owl:unionOf (:PoliticalThing :GeographicalThing).
    # note: Cities are both PoliticalThings and GeographicalThings.

:ReifiedThing a owl:Class;
    owl:disjointUnionOf    ( :Membership
                             :Encompassed
                             :SpokenBy
                             :BelievedBy
                             :EthnicProportion );
    owl:disjointWith :Geometrical.

# the following is not allowed in OWL.
```

```
#:reifies a owl:AnnotationProperty; rdfs:domain :ReifiedRelationship;
#  rdfs:range rdf:Property;
#  owl:inverseOf :reifiedBy.


:name a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain [ a owl:Class;
                  owl:intersectionOf ( :MondialThing
                                       [ owl:complementOf :ReifiedThing ]
                                       [ owl:complementOf :Border ]
                                       [ owl:complementOf :Source ]
                                       [ owl:complementOf :Estuary ]) ];
    rdfs:range xsd:string.

:othername a owl:DatatypeProperty;
    rdfs:domain [ a owl:Class;
                  owl:intersectionOf ( :MondialThing
                                       [ owl:complementOf :ReifiedThing ]
                                       [ owl:complementOf :Border ]
                                       [ owl:complementOf :Source ]
                                       [ owl:complementOf :Estuary ]) ];
    rdfs:range xsd:string.

:type a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain [a owl:Class;
                  owl:intersectionOf ( [owl:unionOf (:GeographicalThing
                                                     :Membership)]
                                       [ owl:complementOf :Continent ]
                                       [ owl:complementOf :River ]
                                       [ owl:complementOf :Source ]
                                       [ owl:complementOf :Estuary ]
                                       [ owl:complementOf :City ]) ];
    rdfs:range xsd:string.

## 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 a owl:Class; owl:unionOf (:Place :Area :Line).
:Location a owl:Class; owl:disjointUnionOf (:Place :SmallArea).
:Area a owl:Class; owl:disjointUnionOf (:SmallArea :LargeArea).
:Place a owl:Class; owl:disjointWith :Area.
:LargeArea a owl:Class; rdfs:subClassOf :Area.
:Line a owl:Class; owl:disjointWith :Area, :Place.

:borders a owl:ObjectProperty;
    rdfs:domain :LargeArea;
    rdfs:range :LargeArea.
```

```
## meta things: measurements
:Measurement a owl:Class.
:TimeSeriesElement rdfs:subClassOf :Measurement.
:year a owl:DatatypeProperty;
    rdfs:domain :TimeSeriesElement;
    rdfs:range xsd:nonNegativeInteger.
:value a owl:DatatypeProperty;
    rdfs:domain :Measurement.

:PopulationCount rdfs:subClassOf :TimeSeriesElement.

####### Political Things
:PoliticalThing a owl:Class;
    owl:disjointUnionOf (:NonGeographicalPoliticalThing :City).
:NonGeographicalPoliticalThing a owl:Class;
    owl:disjointUnionOf (:Country :Province :Organization :Border).
:AdministrativeArea rdfs:subClassOf :Area, :PoliticalThing;
    owl:disjointUnionOf (:Country :Province).
:AdministrativeSubdivision a owl:Class;
    owl:disjointUnionOf (:Province :City).
:PoliticalBody rdfs:subClassOf :PoliticalThing;
    owl:disjointUnionOf (:Country :Organization);
    owl:disjointWith :AdministrativeSubdivision.

:Country rdfs:subClassOf :LargeArea.
:Province rdfs:subClassOf :LargeArea.
:City rdfs:subClassOf :GeographicalThing, :SmallArea.
:Organization a owl:Class; owl:disjointWith :Geometrical.

:carCode a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Country;
    rdfs:range xsd:string.

# a border is a line between two areas (e.g. country/country,
#  sea/sea, country/sea)
:Border rdfs:subClassOf :Line.
:bordering a owl:ObjectProperty;
    rdfs:domain :Border;
    rdfs:range :Country;
    owl:inverseOf :hasBorder.
:hasBorder a owl:ObjectProperty.
:Border rdfs:subClassOf [ a owl:Restriction;
                          owl:onProperty :bordering;
                          owl:cardinality 2 ].

:neighbor a owl:ObjectProperty; a owl:SymmetricProperty;
```

```
    rdfs:subPropertyOf :borders;
    rdfs:domain :Country;
    rdfs:range :Country.

:hasProvince a owl:ObjectProperty; a owl:InverseFunctionalProperty;
    rdfs:domain :Country;
    rdfs:range :Province.
:hasProvince a owl:ObjectProperty;
    owl:inverseOf :isProvinceOf.
:isProvinceOf a owl:ObjectProperty;
    rdfs:subPropertyOf :belongsTo.

:hasCity a owl:ObjectProperty;
    rdfs:domain :AdministrativeArea;
    rdfs:range :City.
:hasCity owl:inverseOf :cityIn.
:cityIn a owl:ObjectProperty.

:City rdfs:subClassOf [a owl:Restriction;
                       owl:onProperty :cityIn;
                       owl:onClass :Country;
                       owl:qualifiedCardinality 1].
:City rdfs:subClassOf [a owl:Restriction;
                       owl:onProperty :cityIn;
                       owl:onClass :Province;
                       owl:maxQualifiedCardinality 1].

:City rdfs:subClassOf :AdministrativeSubdivision.
:AdministrativeSubdivision rdfs:subClassOf [a owl:Restriction;
    ## note: only belongsTo; because locatedIn ist restricted to GeoThings
        owl:onProperty :belongsTo;
        owl:onClass :Country;
        owl:qualifiedCardinality 1].
:City rdfs:subClassOf [a owl:Restriction;
                       owl:onProperty :locatedIn;
                       owl:onClass :Province;
                       owl:maxQualifiedCardinality 1].
:City rdfs:subClassOf [a owl:Restriction;
                       owl:onProperty :locatedIn;
                       owl:onClass :Country;
                       owl:qualifiedCardinality 1].

:population a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain [ owl:intersectionOf ( :Area
                                       [ owl:complementOf :Continent]
                                       [ owl:complementOf :Desert]
                                       [ owl:complementOf :Water]
```

```
                                        [ owl:complementOf :Mountains] )];
    rdfs:range xsd:nonNegativeInteger.
:populationcount a owl:ObjectProperty;
    rdfs:range :PopulationCount.

:capital a owl:ObjectProperty; a owl:FunctionalProperty;
    rdfs:domain :AdministrativeArea;
    rdfs:range :City;
    wl:inverseOf :isCapitalOf.
:isCapitalOf a owl:ObjectProperty.
:City rdfs:subClassOf [a owl:Restriction;
                       owl:onProperty :isCapitalOf;
                       owl:onClass :Country;
                       owl:maxQualifiedCardinality 1].

:populationGrowth a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Country;
    rdfs:range xsd:decimal.
:infantMortality a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Country;
    rdfs:range xsd:decimal.
:gdpTotal a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Country;
    rdfs:range xsd:decimal.
:gdpAgri a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Country;
    rdfs:range xsd:decimal.
:gdpInd a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Country;
    rdfs:range xsd:decimal.
:gdpServ a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Country;
    rdfs:range xsd:decimal.
:inflation a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Country;
    rdfs:range xsd:decimal.
:unemployment a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Country;
    rdfs:range xsd:decimal.
:government a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Country;
    rdfs:range xsd:string.
:independenceDate a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Country;
    rdfs:range xsd:date.
:dependentOf a owl:ObjectProperty; a owl:FunctionalProperty;
    rdfs:domain :Country;
```

```
    rdfs:range :Country.
:wasDependentOf a owl:ObjectProperty; a owl:FunctionalProperty;
    rdfs:domain :Country;
    rdfs:range :PoliticalBody.

:abbrev a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Organization;
    rdfs:range xsd:string.
:established a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Organization;
    rdfs:range xsd:date.
:hasHeadq a owl:ObjectProperty; a owl:FunctionalProperty;
    rdfs:domain :Organization;
    rdfs:range :City.

####### Geographical Things

:GeographicalThing a owl:Class;
    owl:disjointUnionOf (:City :GeographicalNonPoliticalThing).
:GeographicalNonPoliticalThing a owl:Class;
    owl:disjointWith :PoliticalThing;
    owl:disjointUnionOf (:Continent :Water :Source :Estuary :Desert
                         :Island :Mountain :Islands :Mountains).

:Continent rdfs:subClassOf :LargeArea.
:City rdfs:subClassOf :GeographicalThing.  # comment, redundant

:Water a owl:Class; owl:disjointUnionOf (:River :Lake :Sea).
:River rdfs:subClassOf :Line.
:Lake rdfs:subClassOf :SmallArea.
:Sea rdfs:subClassOf :LargeArea.

:Source rdfs:subClassOf :Place;
    owl:disjointWith :Area.
:Estuary rdfs:subClassOf :Place;
    owl:disjointWith :Area.
:Desert rdfs:subClassOf :SmallArea.
:Island rdfs:subClassOf :SmallArea.
:Mountain rdfs:subClassOf :Place.
:Mountains rdfs:subClassOf :SmallArea.
:Islands rdfs:subClassOf :SmallArea.

:area a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain [ owl:unionOf ( :Area :River )];
    rdfs:range xsd:decimal.
:City rdfs:subClassOf [ a owl:Restriction;
                        owl:onProperty :area;
```

```
                          owl:cardinality 0 ].

:length a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Line;
    rdfs:range xsd:decimal.

:elevation a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain [ owl:intersectionOf ( :Location
                                       [ owl:complementOf :Desert] )],
#  muss sein, sonst findet pellet kein range der inversen:
                                    :Location;
    rdfs:range xsd:decimal.

:depth a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain [owl:unionOf (:Sea :Lake)];
    rdfs:range xsd:decimal.

:longitude a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Location;
    rdfs:range xsd:decimal.
:latitude a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Location;
    rdfs:range xsd:decimal.

:mergesWith a owl:ObjectProperty; a owl:SymmetricProperty;
    rdfs:subPropertyOf :borders;
    rdfs:domain :Sea;
    rdfs:range :Sea.

:locatedIn a owl:ObjectProperty;
    rdfs:domain :GeographicalThing,
#  muss sein, sonst findet pellet kein range der inversen ...
                [ owl:intersectionOf ( :GeographicalThing
                    [ owl:complementOf :Continent])];
    rdfs:range :Area.

[ owl:intersectionOf ( :GeographicalThing
                     [ owl:complementOf :Continent])]
    rdfs:subClassOf [ a owl:Restriction;
                        owl:onProperty :locatedIn;
                        owl:onClass :Country;
                        owl:minQualifiedCardinality 1].
:Sea rdfs:subClassOf [ a owl:Restriction;
                         owl:onProperty :locatedIn;
                         owl:allValuesFrom [ owl:unionOf (:Province
                                                          :Country
                                                          :Continent)]].
```

```
:Continent rdfs:subClassOf [ a owl:Restriction;
                             owl:onProperty :locatedIn;
                             owl:allValuesFrom owl:Nothing ].
:River rdfs:subClassOf [ a owl:Restriction;
                         owl:onProperty :locatedIn;
                         owl:allValuesFrom [ owl:complementOf :City]].
:Mountains rdfs:subClassOf [ a owl:Restriction;
                             owl:onProperty :locatedIn;
                             owl:allValuesFrom [ owl:complementOf :City]].
:Desert rdfs:subClassOf [ a owl:Restriction;
                          owl:onProperty :locatedIn;
                          owl:allValuesFrom [ owl:complementOf :City]].
:Lake rdfs:subClassOf [ a owl:Restriction;
                        owl:onProperty :locatedIn;
                        owl:allValuesFrom [ owl:complementOf :Lake]].

# Relationships

:EncompassedArea a owl:Class;
    owl:equivalentClass [ owl:intersectionOf ( :LargeArea
                                               [ owl:complementOf :Continent]
                                               [ owl:complementOf :Sea] )].
:Encompassed rdfs:subClassOf :ReifiedThing.
:encompassedArea a owl:ObjectProperty; a owl:FunctionalProperty;
    rdfs:domain :Encompassed;
    rdfs:range :EncompassedArea;
    owl:inverseOf :encompassedByInfo.
:encompassedBy a owl:ObjectProperty; a owl:FunctionalProperty;
    owl:inverseOf :encompassesInfo;
    rdfs:domain :Encompassed;
    rdfs:range :Continent.

:encompassed a owl:ObjectProperty;
    rdfs:domain :EncompassedArea;
    rdfs:range :Continent;
    owl:inverseOf :encompasses.
:encompasses a owl:ObjectProperty.
:Continent owl:disjointWith :Province.
:Sea owl:disjointWith :Province.

:Membership rdfs:subClassOf :ReifiedThing.

:ofMember a owl:ObjectProperty; a owl:FunctionalProperty;
    rdfs:domain :Membership;
    owl:inverseOf :isInMembership;
    rdfs:range :Country.
:inOrganization a owl:ObjectProperty; a owl:FunctionalProperty;
```

```
    rdfs:domain :Membership;
    owl:inverseOf :hasMembership;
    rdfs:range :Organization.

:isMember a owl:ObjectProperty;
    rdfs:domain :Country;
    rdfs:range :Organization;
    owl:inverseOf :hasMember.
:hasMember a owl:ObjectProperty.

:flowsInto a owl:ObjectProperty;
    rdfs:domain [ owl:unionOf (:River :Lake) ];
    rdfs:range :Water.
:Water rdfs:subClassOf [ a owl:Restriction;
                         owl:onProperty :flowsInto;
                         owl:maxCardinality 1].
:flowsThrough a owl:ObjectProperty; a owl:InverseFunctionalProperty;
#  owl:inverseOf :flowsThrough-; ## das soll nicht explizit benannt sein.
    rdfs:domain :River;
    rdfs:range :Lake.
:hasSource a owl:ObjectProperty; a owl:FunctionalProperty;
    a owl:InverseFunctionalProperty;
    rdfs:domain :River;
    rdfs:range :Source;
    owl:inverseOf :hasSource-.
:hasSource- a owl:ObjectProperty.
:hasEstuary a owl:ObjectProperty; a owl:FunctionalProperty;
    a owl:InverseFunctionalProperty;
    rdfs:domain :River;
    rdfs:range :Estuary;
    owl:inverseOf :hasEstuary-.
:hasEstuary- a owl:ObjectProperty.
:River rdfs:subClassOf [ a owl:Restriction;
                         owl:onProperty :hasSource;
                         owl:cardinality 1],
                       [ a owl:Restriction;
                         owl:onProperty :hasEstuary;
                         owl:cardinality 1].
:Source rdfs:subClassOf [ a owl:Restriction;
                          owl:onProperty :hasSource-;
                          owl:cardinality 1].
:Estuary rdfs:subClassOf [ a owl:Restriction;
                           owl:onProperty :hasEstuary-;
                           owl:cardinality 1].

:locatedAt a owl:ObjectProperty;
    rdfs:domain :City;
```

```
    rdfs:range :Water.
:locatedOnIsland a owl:ObjectProperty;
    rdfs:subPropertyOf :locatedIn;
    rdfs:domain [ owl:unionOf ( :City :Mountain ) ];
    rdfs:range :Island.
:locatedInWater a owl:ObjectProperty;
    # keine rdfs:subPropertyOf :locatedIn, weil river keine area ist
    rdfs:domain :Island;
    rdfs:range :Water.

:belongsToIslands a owl:ObjectProperty; a owl:FunctionalProperty;
    rdfs:domain :Island;
    rdfs:range :Islands.

:Mountain rdfs:subClassOf [ a owl:Restriction;
                            owl:onProperty :locatedOnIsland;
                            owl:maxCardinality 1].

:inMountains a owl:ObjectProperty; a owl:FunctionalProperty;
    rdfs:domain [owl:unionOf (:Mountain :Source)];
    rdfs:range :Mountains.

:Volcano rdfs:subClassOf :Mountain.
:lastEruption a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain :Volcano;
    rdfs:range xsd:date.

# Anthropogeographical Things

:AnthropoGeographicalThing a owl:Class;
    owl:disjointUnionOf (:Language :Religion :EthnicGroup);
    owl:disjointWith :Geometrical.

:Language rdfs:subClassOf :AnthropoGeographicalThing.
:Religion rdfs:subClassOf :AnthropoGeographicalThing.
:EthnicGroup rdfs:subClassOf :AnthropoGeographicalThing.

:AnthropoGeographicalRelationship a owl:Class;
    owl:disjointUnionOf (:SpokenBy :BelievedBy :EthnicProportion).
:SpokenBy rdfs:subClassOf :AnthropoGeographicalRelationship.
:BelievedBy rdfs:subClassOf :AnthropoGeographicalRelationship.
:EthnicProportion rdfs:subClassOf :AnthropoGeographicalRelationship.

#:ofCountry a owl:FunctionalProperty;
#  rdfs:domain :AnthropoGeographicalRelationship; rdfs:range :Country.
:ethnicInfo a owl:ObjectProperty; a owl:InverseFunctionalProperty;
    rdfs:domain :Country;
```

```
    rdfs:range :EthnicProportion;
    owl:inverseOf :ethnicInfo-.
#:ethnicInfo- a owl:ObjectProperty; rdfs:subPropertyOf :ofCountry.
:languageInfo a owl:ObjectProperty; a owl:InverseFunctionalProperty;
    rdfs:domain :Country;
    rdfs:range :SpokenBy;
    owl:inverseOf :languageInfo-.
#:languageInfo- a owl:ObjectProperty; rdfs:subPropertyOf :ofCountry.
:religionInfo a owl:ObjectProperty; a owl:InverseFunctionalProperty;
    rdfs:domain :Country;
    rdfs:range :BelievedBy;
    owl:inverseOf :religionInfo-.
#:religionInfo- a owl:ObjectProperty; rdfs:subPropertyOf :ofCountry.
:onLanguage a owl:ObjectProperty; a owl:FunctionalProperty;
    owl:inverseOf :spokenInInfo;
    rdfs:domain :SpokenBy;
    rdfs:range :Language.
:onReligion a owl:ObjectProperty; a owl:FunctionalProperty;
    owl:inverseOf :followedInInfo;
    rdfs:domain :BelievedBy;
    rdfs:range :Religion.
:onEthnicGroup a owl:ObjectProperty; a owl:FunctionalProperty;
    owl:inverseOf :liveInInfo;
    rdfs:domain :EthnicProportion;
    rdfs:range :EthnicGroup.
:speakLanguage a owl:ObjectProperty;
    rdfs:domain :Country;
    rdfs:range :Language;
    owl:inverseOf :spokenInCountry.
:belongToEthnicGroup a owl:ObjectProperty;
    rdfs:domain :Country;
    rdfs:range :EthnicGroup;
    owl:inverseOf :liveInCountry.
:believeInReligion a owl:ObjectProperty;
    rdfs:domain :Country;
    rdfs:range :Religion;
    owl:inverseOf :believedInCountry.
:percent a owl:DatatypeProperty; a owl:FunctionalProperty;
    rdfs:domain [owl:unionOf (:SpokenBy :BelievedBy
                              :EthnicProportion :Encompassed)];
    rdfs:range xsd:decimal.
```

Appendix C

Meaning of the boolean positions of the RelModelBuilder settings

Position 0 En- or disables the storage of the *Inv* table to the relational database

Position 1 En- or disables the storage of the *MD* table to the relational database

Position 2 En- or disables the storage of the *NMJ* table to the relational database

Position 3 En- or disables the storage of the *NMTables* table to the relational database

Position 4 En- or disables the storage of the *PropTableMap* table to the relational database

Position 5 En- or disables the storage of the *RC* table to the relational database

Position 6 En- or disables the storage of the *AllCl* table to the relational database

Position 7 En- or disables the storage of the *SubCl* table to the relational database

Position 8 En- or disables the storage of the *Keys* table to the relational database

Position 9 En- or disables the storage of the *Hometables* table to the relational database

Position 10 En- or disables the storage of the *AbstractSubclCols* table to the relational database

Position 11 En- or disables the *FindNMTables* step

Position 12 En- or disables the *IdentifyClassTableExtensions* step

Position 13 En- or disables the *IdentifyReifiedTables* step

Position 14 En- or disables the *IdentifyN1Connections* step

Position 15 En- or disables the *DeriveSubclasses* step

Position 16 En- or disables the *FindFKs* step