Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 281

Bootstrapping ontology-based data access specifications from relational databases

Philipp Martis

Course of Study:	Informatik
Examiner:	PD Dr. Holger Schwarz
Supervisor:	M. Sc. Leif Harald Karlsen
Commenced:	23rd of November 2015
Completed:	24th of May 2016
CR-Classification:	D.0, H.2.8

Abstract

Nach der Titelseite des Berichtes und dem Aufgabenblatt soll das Wesentliche aus dem Inhalt der Arbeit in wenigen Sätzen zusammengefasst werden. Diese Übersicht soll keine Formeln und möglichst keine Literaturhinweise enthalten.

Kurzfassung

Nach der Titelseite des Berichtes und dem Aufgabenblatt soll das Wesentliche aus dem Inhalt der Arbeit in wenigen Sätzen zusammengefasst werden. Diese „Übersicht“ soll keine Formeln und möglichst keine Literaturhinweise enthalten.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

As estimated in 2007 [HPZC07], publicly available databases contained up to 500 times more data than the static web and roughly 70 % of all websites were backed by relational databases back then. As hardware has become cheaper yet more powerful, open source tools have become more and more widespread and the web has gotten more and more dynamic and interactive, it's likely that these numbers have even increased since then. This makes the publication of available data in a structured, machine-processable form and its retrieval with eligible software an interesting topic. The most important formalism to represent structured data without the need of a fixed (database) schema is ontologies, and thus this approach is known under the term "Ontology based data access" ("OBDA").

The vision of a machine-processable web emerged as early as 1989 [BL89] and was entitled with the term "semantic web" by Tim Berners-Lee in 1999 [BLF99]. Definitely, the automatic translation of relational databases to RDF [W3C14] or similar representations of structured information is an integral part of the success of the semantic web [HPZC07]. This automatic translation process is commonly called "bootstrapping".

Today, the pure bootstrapping process is a relatively well understood topic, ranging from the rather simple direct mapping approach [W3CR12a] to TODO. On the other hand, the handling of the complexity introduced by these approaches and the use of sophisticated tools to perform various related tasks meanwhile has become a significant challenge in its own right [SGH+15]. Besides the parametrization of the tools in use, this includes the management of the several kinds of artifacts accruing during the process, possibly needed in different versions and formats for the use of different tools and output formats, while also taking changing input data into account [SGH+15]. Skjæveland and others therefore suggested an approach using a declarative description of the data to be mapped, concentrating in one place all the information needed to coordinate the bootstrapping process and to drive the entire tool chain [SGH+15].

1.2. Approach

This thesis describes the development of a specification language to serialize the declarative specification of the bootstrapping process (see Section 1.1 – Motivation) and of a software to in turn bootstrap it from a relational database schema. After the tasks they accomplish, the specification language was called "OBDA Specification Language" ("OSL") and the software bootstrapping the specification was called "db2osl".

Furthermore, this thesis suggests a scheme for generating the IRIs that occur in OBDA spec-

ification, identifying their parts [SGH+15]. Currently, this issue is only exemplified and there is room for improvement in that a simple and straight-forward approach can be used to generate IRIs for all constituents of OBDA specifications without introducing name clashes in corner cases. This approach is described in Section 3.3 – Generating unique IRIs for OBDA specification map fields.

Using a declarative specification makes the entire bootstrapping process a two-step-procedure, illustrated in Figure 1.1: First, the OBDA specification is derived from the database schema using DB2OSL. It specifies the actual bootstrapping process in a very general way, so it only has to be recreated when the database schema changes. The second step is to use the OBDA specification to coordinate and drive the actual bootstrapping process. The development of a software that uses the OBDA specification to perform this second step currently is subject to ongoing work. It will be able to be parameterized accordingly to support different output formats, tools, tool versions and application ranges.

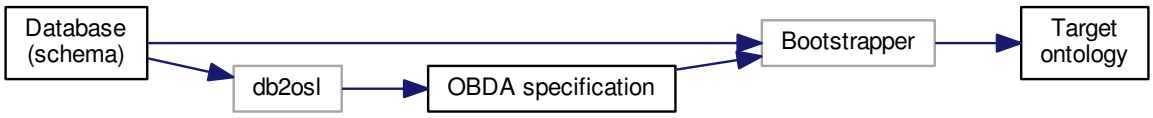


Figure 1.1.: Illustration of the overall bootstrapping process using a declarative OBDA specification

1.3. Requirements and goals

The final system shall be able to cleanly fit into existing bootstrapping systems while being easy to use, taking the burden of dealing with OSL specifications manually from its users instead of adding even more complexity to the process. To achieve these goals, use of existing tools, languages and conventions was made wherever possible. For example, the OBDA SPECIFICATION LANGUAGE was defined to be a subset of OWL. This facilitates meeting the objective of a powerful, easy-to-use, flawless and well-documented language that can be extended and handled by existing tools.

To fit into the environment used in the OPTIQUE project [CGH+13] it is ultimately part of, JAVA was used for the bootstrapping software. Care was taken to design it to be modular and flexible, making it usable not only as a whole but also as a collection of independent components, possibly serving as the basis for a program library in the future. To achieve this aim and to make the software more easily understandable and extensible, it was documented carefully and thoroughly.

As the software will be maintained by diverse people after its development and will likely be subject to changes, general code quality was also an issue to consider. Following good object-oriented software development practice [Str00], real world artifacts like database schemata, database tables, columns, keys, and OBDA specifications were modeled as software objects, provided with a carefully chosen set of operations to manipulate them and make them collaborate. This approach and other actions aiming at yielding clean code are described more thoroughly in Section 6.2 – Code style, while the resulting structure of the software is discussed in Section 5.3 – Architecture of DB2OSL.

2. Background and related work

2.1. Background

2.1.1. Basic concepts

TODO: r2rml, rdf, rdfs, owl, xml, iris, baseiri, end with :

2.1.2. Ontology-based data access (OBDA)

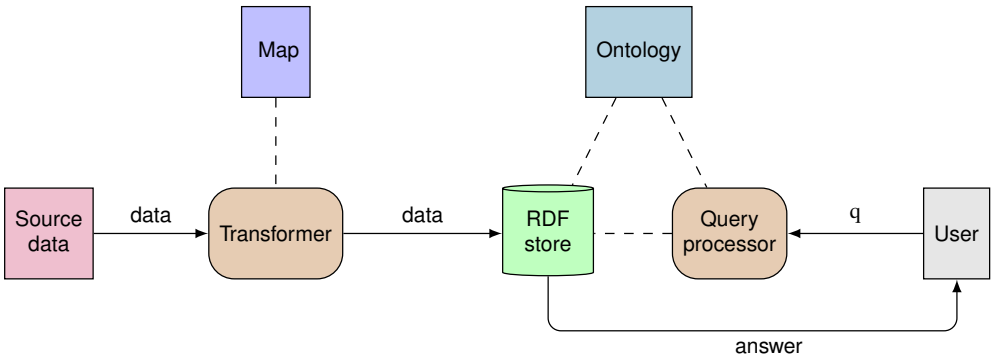
TODO: References

Storing data in relational databases is a very common proceeding, since the notion of a relational database is comprehensible and widely known, while the required software is widely available both commercially and as open-source software. Thus, it is easy for a domain expert to set up and populate a database. Furthermore, relational databases provide significant advantages concerning performance, data consistency and integrity, integration abilities, support and general prominence. These topics definitely played a major role in the success and the extensive exploration that databases discovered, up to the degree that these are the main fields the strengths of databases are seen in. Many – if not all – of these strengths trace back to the relatively fixed and rigid schema databases embody: a well-defined database schema imposing strong and clear cut constraints on the contained data.

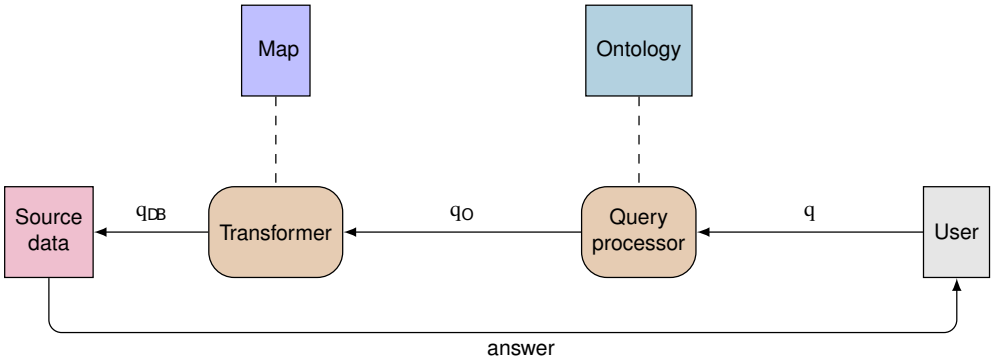
However, this principle also induces notable disadvantages. The database schema, although theoretically changeable, constitutes a significant burden on TODO the representation of data of dynamic environments, incomplete data or changing requirements, especially when dealing with large amounts of data. The resulting representation of data in unintuitive, suboptimal schemata makes the use of prolonged and complex query constructs inevitable, which lets more elaborate queries quickly become unmanageable for non-experts and time-consuming and error-prone even for experts. Ontologies, on the other hand, are much more flexible regarding incomplete data or changing requirements or environments and allow for much more intuitive and abstract query systems, while still being a quite comprehensible formalism (see Section 2.2.1 for publications describing ontologies and the semantic web). Besides, ontologies provide support for different data records referencing the same entity, while databases do not [SGH+15], and for the deduction of implicit information [SGH+15], which with common database systems, if at all, is at least not possible out of the box and in an easy manner. Often, however, relational databases are preferred for their advantages (although the availability of cheaper yet more powerful hardware in some cases offsets these) or simply erroneously. Besides, in some cases, the migration to ontology-based systems, even if beneficial, is to costly to be seriously considered.

Ontology-based data access often provides a solution to this collision of interests: By adding an ontology-based front-end processing the queries that is sensibly mapped to the data representation, the querying facilities of ontology-based systems are introduced, and changes in the data representation most often can be carried out without breaking existing queries; only the mapping has to be changed – in a one-time effort – and only when it introduces changes in the way it presents existing structures to the user, existing queries have to be modified. The creation of these mappings in turn can happen computer-aided or, in simple cases or to a certain degree of completeness and accuracy, the mappings can be completely bootstrapped.

Moreover, in cases where it is to costly or for other cases infeasible to carry out a complete data duplication, the data can remain in the underlying database as is and the query front-end merely acts as an interface transforming the query into a database query [SGH+15] by making use of a backward-chaining technique called *query rewriting* [SGH+15]. This approach is called *virtual OBDA* or *virtual RDF view* [SGH+15] and is illustrated in Figure 2.1b. The oppositional approach of duplicating all data is called *materialized OBDA* or *materialized RDF view* and is illustrated in Figure 2.1a. Virtual OBDA provides limited abilities compared to materialized OBDA in that it does not allow for decoupling from the source data by for example adding inferred information or applying elaborate transformations and does not support "fragments of OWL for which query rewriting is not a complete deduction method" [SGH+15]. However, the response time of systems is hard to predict solely from the architectural approach used, so if it is critical, several systems should be prototyped and evaluated upfront on what are expected to be typical queries [SGH+15].



(a) OBDA system architecture with materialized RDF view



(b) OBDA system architecture with a virtual RDF view

Figure 2.1.: The two basic OBDA system architectures: materialized and virtual RDF views (from [SGH+15])

Finally, it has to be mentioned, that ontology-based data access is not limited to databases. Although this is the most common scenario and the only one this thesis deals with, ontology-based data access also works with other sources of structured information, like ODS, XLS or CSV files, though some additional preparation might be necessary in these cases [SGH+15].

2.1.3. OBDA specifications

TODO: more, maybe shorten introduction

As mentioned in Section 1.1 – Motivation, the sole bootstrapping of RDF triples [W3C14] or other forms of structured information from relational database schemata is a relatively well understood topic. This is outlined more comprehensively in Section 2.2 – Related work.

Nonetheless, bootstrapping remains an elaborate process involving complex tools to be invoked – possibly in different versions and configurations and processing different formats – and working on changing input data [SGH+15]. This is why Skjæveland and others proposed the introduction of OBDA specifications centralizing the task of driving these tools and to gather in one place all the information describing the desired mapping between the source database and the target ontology [SGH+15]. As described in Section 2.2 – Related work, their approach is the foundation of this thesis, which describes and specifies a format for storing and exchanging

such OBDA specifications – the OBDA Specification Language (OSL) – and introduces a tool that in turn automatically bootstraps OBDA Specifications from relational database schemata – the db2osl software.

The bootstrapping process using OBDA specifications and db2osl is illustrated in Figure 1.1 in Section 1.2 – Approach.

2.1.4. The OPTIQUE project

The problems addressed in Section 1.1 – Motivation are a big issue inter alia TODO in the oil and gas industry: 30% to 70% of the working time of engineers is spent on collecting data or assessing its quality [Cro08]. This led to the origination of the OPTIQUE project in TODO which "advocates for a next generation of the well known Ontology-Based Data Access (OBDA) approach to address the data access problem [...] [aiming] at solutions that reduce the cost of data access dramatically" [KGJR+13]. Thus, the OPTIQUE project tries to reach exactly the benefits a well-developed OBDA system can provide (explained in Section 2.1.2): an easy end-user access to data without knowing about its structuring while taking advantage of automatic translations [CGH+13]. In doing so, ascertained shortcomings of existing OBDA systems were addressed: *usability* (for example the need to use formal query languages), *costly prerequisites* (consider, for example, the disadvantages of materialized OBDA described in Section 2.1.2) and *efficiency* (which was perceived as being insufficiently addressed in previous approaches) [KGJR+13].

2.2. Related work

2.2.1. Ontologies and the semantic web – publications

2.2.2. OBDA specifications – publications

A publication building the foundation of the work presented in this thesis, is the summarizing and benchmarking work on OBDA specifications by Skjæveland et al. [SGH+15], the group that developed them in their present form.

2.2.3. OBDA systems – publications

2.2.4. General ontology bootstrapping – publications

Skjæveland, Lian and Horrocks [SLH13] provided an exemplifying description of the transformation of the *NPD FactPages*, an enormous collection of data related to oil drilling on the Norwegian continental shelf, provided by the Norwegian Petroleum Dictorate (NPD).

Sequeda et al. [STCM11] provided an overview over different direct mapping approaches.

Sequeda, Arenas and Miranker [SAM12] [SAM11] describe the direct mapping of relational databases to RDF and OWL formally.

Stojanovic, Stojanovic and Volz [SSV02] published a formal description of the mapping of relational databases onto ontology-based structures, describing concepts preceding and/or supplementing OWL and using F-LOGIC TODO as target language.

2.2.5. The OPTIQUE project – publications

Calvanese et al. [CGH+13] presented the OPTIQUE project including its underlying OBDA system and showed limitations of current OBDA systems.

Kharlamov et al. [KGJR+13] described the first version of the OPTIQUE system, customized for use with the *NPD FactPages* of the Norwegian Petroleum Dictorate (NPD).

Skjæveland and Lian [SL13] summarized the benefits of and the proceeding for converting the NPD FactPages to linked data [BHBL09] and discuss associated terms like linked data, URIs, RDF and SPARQL.

2.2.6. Alternative approaches – publications

Barrasa, Corcho and Pérez [BRCGP04] proposed a declarative mapping language – R2O – able to express a mapping between a relational database and ontologies represented in the OWL and RDF formats. This approach however aims at connecting existing databases and existing ontologies.

TODO: R2RML, SQL2SW

3. On bootstrapping and IRI generation

3.1. Ontology bootstrapping using direct mapping

TODO: more, individuals <- data TODO: no alternative approach

As its name suggests, the direct mapping approach is a relatively simple and straight forward approach. Direct mapping is currently a W3C recommendation, which defines the production of an RDF graph TODO – which is called the *direct graph* – from a relational schema [W3CR12a]. As a matter of fact, the main definition of the direct graph, excluding definitions of rather trivial subcomponents, fits on one computer screen.

The direct graph contains all data held in the source database but it does not contain additional schema information like uniqueness of or non-null constraints on columns [W3CR12a].

3.1.1. Overview on the direct graph

The constitution of the direct graph is illustrated in Figure 3.1. Its basic components are, for each row, the *row type triple*, its *literal triple*s and its *reference triple*s. Here, the row type triple encodes which table the respective row belongs to, the literal triples encode the data in non-foreign-key columns and the reference triples encode the data in foreign key columns. These triples are then by degrees united to the direct graph: the row triples of each row form the row graph, the row graphs of each table form the table graph and all table graphs united constitute the final direct graph.

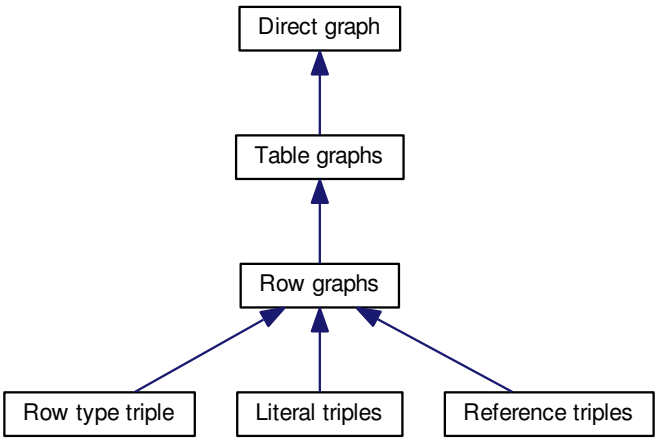


Figure 3.1.: Constitution of the *direct graph*. "→" means "is part of".

Carefully assigning IRIs to the RDF entities is an essential part of the approach, since oth-

erwise, name clashes can occur. Indeed, there seems to be a corner case which was not considered. For details on IRI generation in direct mapping, see Section 3.1.3.

3.1.2. Data representation in direct mapping

Since the result of a direct mapping is an RDF graph, the means to represent data are limited to valid RDF vocabulary. This is no problem for IRIs and expressions that only involve IRIs (row type triples and reference triples).

To encode the non-foreign-key data content of the source database, thus literal triples, the R2RML mapping language is used, a language providing a mapping from the relational data model to the RDF data model [W3CR12b]. R2RML expressions thereby are by themselves RDF statements. The data value contained in a direct mapping literal triple is defined to be the *R2RML natural RDF literal* representation of the value [W3CR12a], which, as the name suggests, is a single RDF literal [W3CR12b].

3.1.3. IRI generation in direct mapping

As all RDF triples generated by the direct mapping are simply united to constitute the final *direct graph* (see Section 3.1.1 – Overview on the direct graph for details), a senseful IRI assigning is vital to the functioning of the approach. By design, IRIs for different kinds of entities have a different structure, which prevents name clashes on the one hand, but on the other hand induces that, in case of a clash, all entities with the conflicting IRI are of the same kind, which means a high risk of producing ambiguous information and thus losing data.

The relatively simple way IRIs are assigned in direct mapping is described in the following [W3CR12a]:

- Table IRIs correspond to the table name.
- Literal property IRIs consist of the table name and the column name, separated by a hash character ('#').
- Reference property IRIs consist of the child table name, the string "#ref-" and the child table column names of the respective foreign key, separated by a semicolon (';').
- All contained names are included in their percent-encoded form TODO.

The encoding of reference property IRIs can lead to name clashes in cases multiple foreign keys exist which contain exactly the same columns – which is allowed for example in SQL TODO. To remove this flaw, the parent table name and the parent table column names of the respective foreign key must also be included in the IRI.

3.2. Ontology bootstrapping using OBDA specifications

TODO: only mapping, no duplication, r2rml

3.2.1. Structure of OBDA specifications

An OBDA specification consists of several so-called "maps", which are data records containing data and references to each other describing parts of the OBDA specification in statically defined fields [SGH+15]. For different aspects of the specification, there are different map types, while usually several maps exist for each type. Namely, these are *Entity maps* describing database tables, *Identifier maps* describing database primary keys, *Attribute maps* describing database columns, *Relation maps* describing database foreign keys, *Subtype maps* describing "is-a" relationships in the data and *Translation tables* describing desired translations of data.

The fields of the several types of maps and their interconnection via references is shown in Figure 3.2. Here, each field specifies, in that order, the field label, the field's long name, the bootstrapping steps in which the field is used and the field's short name. Fields storing a set of values have both their short and their long name suffixed with "...". Note that each reference between two fields is denoted with a short field name contained in the source of the reference, specifying the field in which the reference is stored. What the values of the fields of the several types of maps express exactly and how they can be used is described in Section 3.2.2. For a full description as well of the structure of OBDA specifications as of their application and the general idea behind them, refer to [SGH+15]. How OBDA specifications can in turn be automatically bootstrapped, excluding Subtype maps and Translation tables is described in Section 3.2.3.

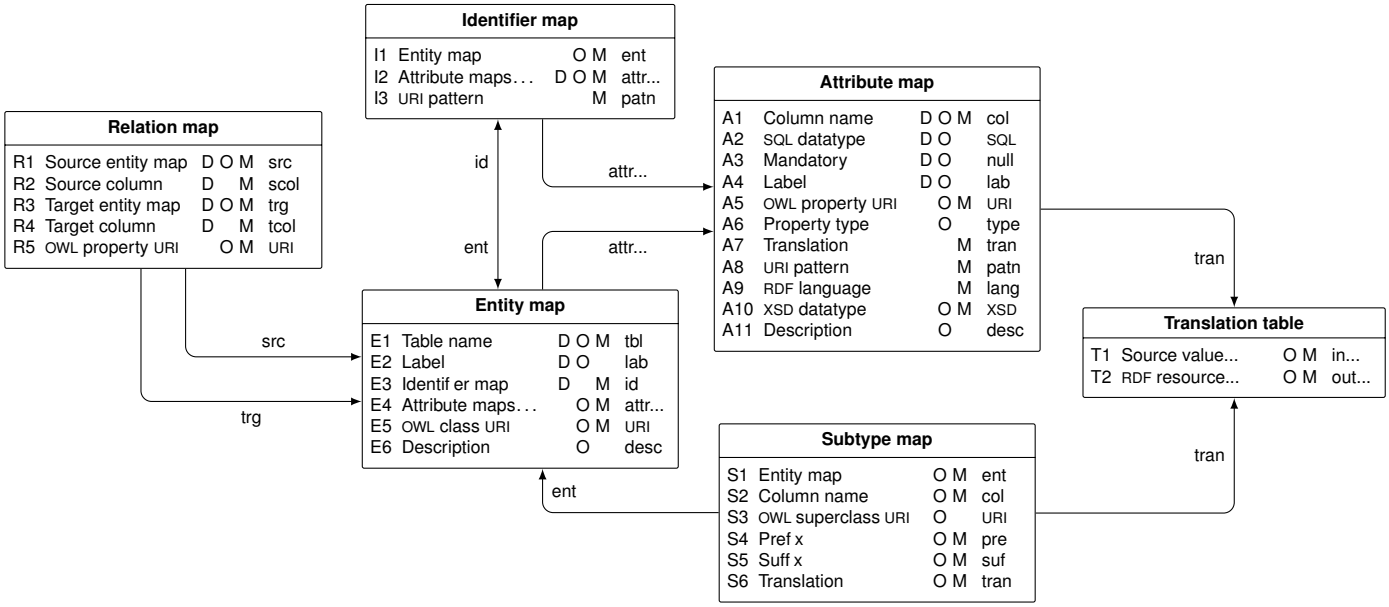


Figure 3.2.: Constitution of an OBDA specification. "→" means "references". (from [SGH+15])

Entity maps, Identifier maps, Attribute maps and Relation maps directly relate to database concepts and each of them describes exactly one database table, primary key, column or foreign key, respectively, and vice versa. Subtype maps and Translation tables, on the other hand, represent concepts of the bootstrapping process or data to be added to the target ontology and are somewhat harder to obtain: Subtype maps represent "is-a" relationships in the target ontology to be determined from the source data [SGH+15] – heuristically or semi-automatically

–, while Translation tables allow for the transformation of data values, for example from `TRUE` to `true` or from `No` to `false` [SGH+15]. Therefore, they also have to be determined heuristically or semi- automatically from the source data – considering the database schema only is not sufficient [SGH+15]. Note that, because of this, special care has to be taken to keep the maps synchronized with the data in case of Subtype maps and Translation tables.

The structural description of OBDA specifications in [SGH+15] does not propose a serialization format in which OBDA specifications can be stored or read and written by software and human agents. How this can be done is subject to this thesis, which introduces the OBDA Specification Language (OSL) designed exactly for this purpose in Chapter 4.

3.2.2. Using OBDA specifiations

TODO: dirm TODO: r2rml

As described in in Section 2.1.3 – OBDA specifications, using OBDA specifications provides several benefits when concerned in ontology bootstrapping. Principally, information about the bootstrapping process is collected in one place and can be used to manage the tools involved. This includes the availability of the URIs to be used in the constructed ontology from a central place, which is a great advantage, since URIs are central to an ontology TODO. Additionally, all information on the database schema of the source database is available. Using Translation tables, all these information can at will be made subject to transformations normalizing or correcting the data changing the database [SGH+15].

By the use of a single specification language like the OBDA Specification Language (OSL) to store OBDA specifications, the expense of converting between different data formats can be reduced significantly: assumed that there are n different formats to be handled with no means provided to convert between them, the converting costs decrease from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ by introducing a single central language TODO.

Besides the structure of OBDA specifications, described in Section 3.2.1, Skjæveland et al. introduce a set of formal rules defining the bootstrapping process and the mapping of the source data to the generated ontology [SGH+15]. Using an OBDA specification, tools implementing these rules can easily and in a well-defined manner bootstrap an ontology and a mapping from the source data onto this ontology (or duplicate that source data, if the materialized OBDA approach is used, see Section 2.1.2). The mapping rules produce RDF triples which can be interpreted by R2RML to establish the mapping (see Section 2.1.1 – Basic concepts). They are accompanied by SQL statements specifying the queries over the source database used to link the ontology to the data. If the data source is not a relational database but another form of structured data, like CSV files, a database schema can be bootstrapped first by applying additional "database rules" [SGH+15]. Afterwards, the proceeding can continue as if the data source were a database, so this case is neglected in the following.

The rest of this section contains description of the information contained in the various types of OBDA specification maps and how it is used during bootstrapping, based on the description in [SGH+15]. Keep in mind that "bootstrapping" here refers to the ontology bootstrapping process specified by the OBDA specification, yielding a target ontology and mappings relating it to the source database – it does not refer to the bootstrapping of the OBDA specification itself. The text is meant to give a brief explanatory overview over the bootstrapping process

using OBDA specifications. Thus, it focuses on the information they provide and how they are used to link the bootstrapped ontology to the source data, leaving out the SQL statements to be used to gain the datasets. How exactly the ontology is created is also left out, since this would have involved introducing too many technical details, and moreover, the topic is also comprehensible by describing the mapping only. For a detailed description of the bootstrapping process, including ontology creation and all formal rules to be applied, see [SGH+15]. For an explanation about how an OBDA specification containing Entity maps, Identifier maps, Attribute maps and Relation maps can be bootstrapped from a relational database schema, see Section 3.2.3. For details on URI generation, refer to Section 3.3 – Generating unique IRIs for OBDA specification map fields.

Entity maps

Entity maps provide information about the tables contained in the source database or in the intermediate database schema to be constructed, if the data source is not a database but some other source of structured information (see Section 2.1.2 – Ontology-based data access (OBDA)). The information provided by an Entity map includes the table name, a label describing the table and a (more detailed) description of the table. Furthermore, each Entity map references an Identifier map representing its primary key and a set of Attribute maps representing its columns. Finally, an Entity map provides an OWL class URI identifying the represented table uniquely in the resulting ontology. As the name suggests, this URI is given to an OWL class which serves as OWL type (or more precisely: `rdf:type`) for all OWL individuals representing the datasets from the respective table in the target ontology (see Paragraph "Identifier maps").

Suppose, for example, that an Entity map for a table "`persons`" provides the OWL class URI "`mydb:persons`". Then, in the target ontology, all OWL individuals representing rows in the "`persons`" table, will be of `rdf:type` "`mydb:persons`". If a data record in the "`persons`" table has the identifying URI pattern "`mydb:person/{pno}`" (see Paragraph Identifier maps), this type information will be expressed by the following RDF triple:
`mydb:person/{pno} rdf:type mydb:persons`.

Identifier maps

Identifier maps describe database primary keys contained in the source database or in the intermediate database schema to be constructed, if the data source is not a database (see Section 2.1.2). Each Identifier map contains a reference back to the respective Entity map, representing the database table the primary key belongs to. Furthermore, each Identifier map references a set of Attribute maps, representing the database columns the primary key consists of (see next paragraph). Finally, an Identifier map provides a URI pattern, allowing OWL individuals in the bootstrapped target ontology that represent datasets to be identified. A URI pattern contains placeholders like "`{$1}`" for all primary key columns, which are replaced with the respective column names, surrounded by curly braces, during the bootstrapping process, to yield a valid R2RML template [W3CR12b]. Since the column name substituted in is still a placeholder, the result of this substitution is still a URI pattern and not a URI. Since such a URI pattern uniquely identifies a dataset from a given database table – when data values are

substituted in –, it will be called *identifying URI pattern* in the following.

Consider the URI pattern "`mydb:person/{$1}`". Replacing the placeholder "`{$1}`" with the column name "`pno`" in curly braces yields the following identifying URI pattern: "`mydb:person/{pno}`".

Attribute maps

Attribute maps provide information about database columns contained in the source database. Each Attribute map carries the column name, information whether having a value in this column is mandatory for a dataset (in SQL terms: whether it has a `NOT NULL` constraint), a label describing the column as well as an extended description of the column. A database column is represented as a relation in the final ontology, thus, in OWL terms, as an `owl:DataProperty` or an `owl:ObjectProperty`. The Attribute map provides the URI for this OWL property.

Additionally, it specifies the datatype of values in the represented column in the following manner: Three fields are provided for this purpose, `SQL datatype`, `RDF language` and `XSD datatype`. If the `XSD datatype` field is nonempty, its value is specified to be the datatype for values in the column the Attribute map represents (note that OWL only knows XSD datatypes TODO). Otherwise, if the value of the `SQL datatype` is a standard SQL type, it will be mapped to an XSD datatype and the resulting type is specified as datatype for values in the respective column. If neither of the above is the case and the `RDF language` field is nonempty, values in the respective column will be interpreted as strings with the value of the `RDF language` field applied as RDF language tag (TODO). If neither of the above is the case, values in the respective column will be interpreted as strings without an RDF language tag.

Finally, an Attribute map specifies whether the column shall be represented as an `owl:DataProperty` (for non-foreign-key columns) or as an `owl:ObjectProperty` (for foreign key columns). The field specifying that – `Property type` – also allows, as a further distinction of object properties, whether the property's target URI should be the column name placed in an URI pattern provided by the Attribute map, similarly to URI patterns in Identifier maps (see example at the end of this paragraph), or if it shall simply be the column name, possibly with a translation from a Translation table, specified by the Attribute map, applied. The former option is useful for example when using custom property URIs to express relations between source data and the target ontology. If an `owl:DataProperty` is generated, it always has the column name as target URI, without the use of an URI pattern. Note that it is sufficient to have the column name be the target of the property, since only a *mapping* to the source data is generated.

Consider an Attribute map representing a column named "`name`" to be mapped to an `owl:DataProperty`. Suppose the OWL property URI the Attribute map specifies is "`mynamespace:lastName`" and each dataset containing the `name` column has the identifying URI pattern "`mydb:person/{pno}`" (see Paragraph "Identifier maps"). Then, during the bootstrapping process the following RDF triple will be produced:
`mydb:person/{pno} mynamespace:lastName "{name}"`.
This triple can easily be interpreted by R2RML, which on request then retrieves the queried `name` from the data source.

Consider, as a more elaborate example, an Attribute map representing a column named

"company" and to be mapped to an owl:ObjectProperty with the use of the URI pattern "http://otherdb/{$1}". Suppose the OWL property URI the Attribute map specifies is "mynamespace:hasSameOwnerAs", there is no datatype specified and each dataset containing the company column has the identifying URI pattern "mydb:company/{cmpno}" (see Paragraph "Identifier maps"). Then, the following RDF triple will be produced during the bootstrapping process:
mydb:company/{cmpno} mynamespace:hasSameOwnerAs http://otherdb/{company}.
Note that this only makes sense if R2RML can expand "http://otherdb/{company}" to a valid subject for each value in the company column of the database, and if all rows in the respective database table are indeed entities having the same owner as the company specified in the company column.

Relation maps

Relation maps represent foreign keys contained in the source database. Each Relation map references the Entity maps representing the foreign key's child table and parent table, respectively. Furthermore, it provides the column names of both the foreign key columns and the referenced columns and specifies an OWL property URI. Relation maps allow the relations expressed in the source data via foreign key relationships to be included into the bootstrapped ontology. This happens in a simple and straight-forward manner: for each foreign key relationship, exactly one triple is generated which contains the two identifying URI patterns representing the source and the target dataset of the foreign key, respectively, and the OWL property URI specified by the Relation map.

Suppose, for example, that a Relation map expressing a relation between datasets with the identifying URI patterns (see Paragraph "Identifier maps") "mydb:persons/pno/{pno}" and "mydb:companies/cmpno/{cmpno}", respectively, and specifying the OWL property URI "mynamespace:isEmployedAt". This will result in the following RDF triple to be generated during the bootstrapping process:
mydb:persons/pno/{pno} mynamespace:isEmployedAt mydb:companies/cmpno/{cmpno}

Subtype maps

Subtype maps provide a means to automatically add subclass-superclass relationships to the target ontology during the bootstrapping process. They specify an Entity map and a column name defining a table and a column, respectively, that exist in the source database and contain the values to be declared as belonging to the subclass. Furthermore, they store a prefix, a suffix and possibly a reference to a Translation table which are used to generate a URI for that subclass. Finally, they provide the URI of the superclass. This can be, for instance, some OWL class being created during the bootstrapping process or already existing in some imported ontology. The URI generated for the subclass contains the data value of the respective database column, thus every dataset gets its own (sub)class. During bootstrapping, an RDF triple declaring the value to belong to that subclass is generated for each data value of the respective table column in the source database. The limitation to the desired value only thereby happens by a restriction on the SQL statement accompanying the respective triple, not by limiting the triple to only cover a specific data value. The actual subclass-superclass relationship

is expressed during the creation of the ontology. Note the difference from the previously described mapping rules, which produced triples independent from the data values in the source database.

Consider a Subtype map specifying a table column containing the values "`Purchase`" and "`Sales`" with the datasets having the identifying URI pattern (see Paragraph "Identifier maps") "`mydb:managers/mno/{mno}`". Suppose, the Subtype map specifies the prefix "`mydb:manager/of_department/`", no suffix, no translation table and the supertype URI "`mynamespace:persons`". This will result in the generation of the following triples during the bootstrapping process:
`mydb:managers/mno/{mno} rdf:type mydb:manager/of_department/Purchase`
`mydb:managers/mno/{mno} rdf:type mydb:manager/of_department/Sales`,
while "`mydb:manager/of_department/Purchase`"
and "`mydb:manager/of_department/Sales`" will be subclasses of class
"`mynamespace:persons`" in the target ontology. The accompanying SQL statement will ensure that, despite the use of the R2RML template "`mydb:managers/mno/{mno}`", not *every* manager will be declared as the manager of *every* department.

Translation tables

Translation tables allow for transforming URIs or other strings in arbitrary ways, by simply mapping each string to be translated to a target string.

They don't reflect in the target ontology in any form but are used only during the bootstrapping process.

3.2.3. Bootstrapping OBDA specifications

How OBDA specifications can in turn be bootstrapped from database schemata is subject to this thesis and is explained in this section. For the description of the software developed to automate this, see Chapter 5 – The DB2OSL software. The description in this section assumes an SQL database as data source. However, ontology-based data access and OBDA specifications are not limited to SQL databases, as mentioned in Section 2.1.2 – Ontology-based data access (OBDA) and in Section 3.2.2 – Using OBDA specifiations. Furthermore, this section refers to OBDA specifications without assuming any specific format in which they are represented. How OBDA specifications are represented internally by the DB2OSL software, is described in Section 5.3.3 – Fine structuring of DB2OSL. For the description of a format to serialize OBDA specifications – the output format of DB2OSL –, refer to Chapter 4 – The OBDA Specification Language (OSL).

Subtype maps and Translation tables are not considered in this approach, since they cannot be bootstrapped from schema information only but have to be determined from the input data (see Section 3.2.1 – Structure of OBDA specifications). Thus, the bootstrapped OBDA specification does not contain maps of these types. Including them is a significant challenge in its own right and, since the use of heuristics or user decisions would be necessary, would make the process involve human supervision at least. Apart from that, the bootstrapping is an easy and straight-forward task which can be carried out fully automatic TODO.

Recall the structure of an OBDA specification explained in Section 3.2.1 – Structure of OBDA specifications. The map types considered in this approach are Entity maps, Attribute maps, Identifier maps and Relation maps. The assignment of values to their fields is summarized in Table 3.1, only hinting at how maps are generated. Both the generation of the maps and the assignment of values to their fields are described in the rest of this section, with one exception: since the generation of URIs (or IRIs) in the context of OBDA specifications is an essential topic which requires some conceptual efforts, it is described in a separate section, Section 3.3.

Map type	**Field name**	**Value**
Entity map	Table name	SQL table name
Entity map	Label	<empty>
Entity map	*Identifier map*	Identifier map for table
Entity map	*Attribute maps...*	Attribute maps for table columns
Entity map	OWL class URI	URI(table)
Entity map	Description	SQL table description
Identifier map	*Entity map*	Entity map for corresponding table
Identifier map	*Attribute maps...*	Attribute maps for primary key columns
Identifier map	URI pattern	URIpattern(table)
Attribute map	Column name	SQL column name
Attribute map	SQL datatype	SQL datatype of column
Attribute map	Mandatory	SQL `NOT NULL` property of column (`true` or `false`)
Attribute map	Label	<empty>
Attribute map	OWL property URI	<empty> for foreign key columns, else URI(table, column)
Attribute map	Property type	"`ObjectProperty`" for foreign key columns, else "`DataProperty`"
Attribute map	*Translation*	<empty>
Attribute map	URI pattern	<empty>
Attribute map	RDF language	<empty>
Attribute map	XSD datatype	<empty>
Attribute map	Description	SQL column description
Relation map	*Source entity map*	Entity map for foreign key child table
Relation map	Source column	Foreign key child columns (SQL column names)
Relation map	*Target entity map*	Entity map for foreign key parent table
Relation map	Target column	Foreign key parent columns (SQL column names)
Relation map	OWL property URI	URI(table, foreignKey)

Table 3.1.: Assignment of values to fields of OBDA specification maps

Entity maps

Exactly one Entity map and one Identifier map is generated per table contained in the source database. The generated Identifier map is referenced by the Entity map's `Identifier map` field. Similarly, exactly one Attribute map is generated per table column and these Attribute

maps are referenced by the Entity map's `Attribute maps...` field. The Entity map's `Table name` field is set to the SQL name of the table, the `Label` field remains empty. An URI identifying the table is generated and stored in the Entity map's `OWL class URI` field. The SQL table description is copied into the Entity map's `Description` field.

Identifier maps

An Identifier map represents exactly one primary key in the source database and is referenced by the Entity map representing the table containing the primary key constraint. In addition, it references this table in its `Entity map` field, so that there is a bidirectional referencing. The Attribute maps representing the columns constituting the primary key are referenced by the Identifier map's `Attribute maps...` field. An URI pattern, allowing datasets (thus, rows in the source database) to be identified in the target ontology, is generated and put in the Identifier map's `URI pattern` field.

Attribute maps

An Attribute map represents exactly one column in the source database and is referenced by the Entity map representing the table containing the column. The Attribute map's `Column name` field is set to the SQL column name of the column, the `SQL datatype` field is set to its SQL datatype. The `Mandatory` field is set to `true` if the column has the SQL `NOT NULL` constraint, otherwise to `false`. If the column is part of a foreign key, the `OWL propert URI` field remains empty. Otherwise, an URI identifying the column is generated and stored in the Attribute map's `OWL property URI` field. The `Property type` field is set to "`ObjectProperty`" if the column is part of a foreign key, otherwise to "`DataProperty`". The SQL column description is copied into the Attribute map's `Description` field. The remaining columns, `Label`, `Translation`, `URI pattern`, `RDF language` and `XSD datatype` remain empty.

Relation maps

A Relation map represents exactly one foreign key in the source database. It contains fields storing the parent and child table of the foreign key: the `Source entity map` field, referencing the Entity map representing the child table of the foreign key, and the `Target entity map` field, referencing the Entity map representing the parent table of the foreign key. The SQL column names of the foreign key columns (thus, column names in the child table) are copied into the `Source column` field of the Relation map, and the SQL column names of the referenced columns in the parent table are copied into its `Target column` field. Note that, in contrast to Identifier maps representing primary keys, it is not referenced by any Entity map (or any other map).

3.3. Generating unique IRIs for OBDA specification map fields

As explained in Section 2.1.1 – Basic concepts, IRIs play a central role in diverse topics related to ontology-based data access. They provide the means to uniquely identify entities, which of course is a necessity for data retrieval. As also explained in Section 2.1.1, every URI is also a IRI, so although Skjæveland et al. use the term "URI" in the introduction of their approach of using OBDA specifications for ontology bootstrapping – and that term is also used in Section 3.2, which describes this approach – in this section the general term "IRI" is used, marking that the introduced concepts are valid for all types of IRIs.

When dealing with ontology bootstrapping using OBDA specifications, it is important to differentiate between the three types of IRIs occurring in this matter, which will be underlined by the following unambiguous naming:

- *Data IRIs* identify entities in the bootstrapped ontology
- *OBDA IRIs* are used as values for the fields of OBDA specification entities
- *OSL IRIs* identify components in serialized OBDA specifications, using the OBDA Specification Language (OSL) introduced in Chapter 4 for serialization

Skjæveland et al. do not define or assume a particular scheme for IRI generation in their introduction of OBDA specifications [SGH+15]. Instead, the IRI generation strategy is only adumbrated by giving examples of entities having IRIs. The examplified scheme was used for the implementation of the db2osl software bootstrapping OBDA specifications from relational database schemata, which is described in this thesis (see Section 5 – The db2osl software and Section 6 – Implementation of db2osl). The direct mapping approach for ontology bootstrapping described in Section 3.1, on the other hand, introduces a scheme for IRI generation [W3CR12a], but with this scheme, name clashes can occur, as explained in Section 3.1.3. The OBDA Specification Language (OSL), finally, defines a proper scheme for OSL IRIs, as is explained in Section TODO.

In the following, an enhanced scheme for the generation of OBDA IRIs is proposed, which resembles the previously mentioned scheme used for OSL IRIs and which also may serve as a blueprint for other IRI generation strategies.

3.3.1. Requirements for the IRI scheme

As explained in Section 2.1.1 – Basic concepts, the main requirement on a IRI generation scheme is uniqueness of the IRIs: no two entities must be possibly assigned the same IRI, regardless of their kind, of how low the probability of a name clash (IRI collision) is or of the conditions leading to a name clash. Additionally, IRI uniqueness shall be independent from the base IRIs, thus a base IRI shall be arbitrarily selectable for each generation process without introducing name clashes even with IRIs having other base IRIs.

As to OBDA specification entities, the following kinds of IRIs have to be available, including IRI patterns:

- Entity map OWL class IRIs

- Identifier map IRI patterns
- Attribute map OWL property IRIs
- Attribute map IRI patterns
- Relation map OWL property IRIs
- Subtype map (IRI) prefixes
- Subtype map (IRI) suffixes
- Subtype map OWL superclass IRIs

As Subtype map OWL superclass IRIs are IRIs of data entities already existing in the target ontology by some means (see Section 3.2.2 – Using OBDA specifiations), they do not have to be generated and thus are ignored in the following. Exactly the same holds for Attribute map IRI patterns. Furthermore, this approach creates Subtype map IRI prefixes already leading to unique IRIs for Subtype map subclasses and so Subtype map IRI suffixes are ignored in the following. Since an IRI generation scheme cannot avoid collisions with existing IRIs out of its outreach and these collisions can easily be prevented, for example, by giving them another base IRI (see Setion 2.1.1 – Basic concepts), this case is excluded from the requirement that no two URIs must collide under any circumstances. However, the user shall be able to chose such externally generated IRIs from an infinite set of IRIs, while being sure that no name clashes will occur.

So compendious, the requirements on the IRI generation scheme are that Entity map OWL class IRIs, Identifier map IRI patterns, Attribute map OWL property IRIs, Relation map OWL property IRIs and Subtype map IRI prefixes can be generated that, regardless of the chosen base IRIs, don't clash among another, while leaving an infinite set of predictable IRIs that don't clash with any of the generated IRIs.

3.3.2. Avoiding name clashes in the IRI scheme

Generating unique Entity map OWL class IRIs ignoring base IRIs is not much of a problem, assuming database table names are distinct, which is guaranteed in a common database system like SQL [**sql**]. Including the table name into an Entity map OWL class IRI is sufficient to prevent it from colliding with other IRIs with the same base IRI. However, when taking two different base IRIs into account that are used for two IRIs created according to this scheme, things get more complicated.
Consider, for example, a database table named "`Persons`" and a table named "`Persons__TABLE__Persons`". Generating an IRI according to the scheme "<base:>`TABLE__`<table name>" for each of these tables, using the base IRI "`TABLE__Persons__`" for the first one and the empty base IRI for the second one, both tables will get the IRI "`TABLE__Persons__TABLE__Persons`", although the table name was included into the IRI in both cases. The problem is that the "`TABLE__`" string occurring in the table name cannot be discriminated from the "`TABLE__`" string added in the course of IRI generation or the "`TABLE__`" string occurring in the base IRI. To solve the problem, a marker has to be included in the URI which definitely indicates the beginning of the table name. In addition, this marker will uniquely identify Entity map OWL class IRIs. For both aims to be achieved, an escape symbol must be used, which makes the marker unique at least outside the base IRI part, by

escaping the marker whenever it occurs in the table name.

Regarding Identifier map IRI patterns, the IRI resulting from the expansion of the pattern will contain the column names of the primary key represented by the respective Identifier map [SGH+15]. Further on, the table name of the database table containing that primary key has to be included in the IRI pattern, since two distinct tables may have primary keys with equally named columns. This will make the IRI pattern a unique Identifier map IRI pattern, since a database table can be assumed to only have one primary key, as is the case in common database systems like SQL [**sql**]. The fact that primary key values are unique for each dataset ensures that unique Identifier map IRI patterns expand to unique IRIs. Moreover, it has to be ensured, that IRIs resulting from the expansion of Identifier map IRI patterns do not collide with IRIs of other kinds.
Taking arbitrary and particularly varying base IRIs into account, a definite marker has to be included in the IRI pattern and other occurrences of this marker in the IRI pattern have to be escaped. This uniquely identifies Identifier map IRI patterns and unambiguously distinguishes the table name from the rest of the IRI.

Concerning Attribute map OWL property IRIs, they will be unique among their kind when they include the column name of the database column they represent besides the table name of the table containing it, since database table names can be assumed to be distinct and column names can be assumed to be unique within a table, which is guaranteed in a common database system like SQL [**sql**]. Furthermore, Attribute map OWL property IRIs have to be prevented from colliding with IRIs of other kinds.
Taking arbitrary and particularly varying base IRIs into account, definite markers have to be included in the IRI and other occurrences of this marker in the IRI have to be escaped. This uniquely identifies Attribute map OWL property IRIs and unambiguously distinguishes the table name and the column name from the rest of the IRI and from one another.

Regarding Relation map OWL property IRIs, including the table name and the column names of both the foreign key represented by the Relation map (or the containing table, respectively) and the referenced key (or its containing table, respectively) in the IRI will make it a unique Relation map OWL property IRI. Note that including only the table name and the column names of the foreign key (or its containing table, respectively) would not be sufficient, since several distinct foreign keys covering exactly the same columns can exist in a table (this is what the IRI generation scheme of the direct mapping approach misses). The same applies of course for the referenced table and its columns – several foreign keys can reference them. Moreover, these Regarding Relation map OWL property IRIs have to be prevented from colliding with IRIs of other kinds.
Taking arbitrary and particularly varying base IRIs into account, definite markers have to be included in the IRI pattern and other occurrences of this marker in the IRI have to be escaped. This uniquely identifies Relation map OWL property IRIs and in particular their parts providing the table and column names.

Concerning Subtype map IRI prefixes, they must include the column name of the database column containing the values to be declared belonging to the subclass. Further on, since another database table could contain a column of the same name, the IRI must include the table name of the database table containing the column. This will make Subtype map IRI prefixes unique among their kind. Note that a Subtype map IRI prefix, similarly to a IRI pattern, does not specify the final IRI but is subject to expansion. This expansion can yield

the same IRI for different data records, which, however, is not considered a collision, since this behavior is intentional – every two data records having the same value in the respective column, and only those, will get the same IRI. Additionally, it has to be ensured, that IRIs resulting from such an expansion do not collide with IRIs of other kinds.
Taking arbitrary and particularly varying base IRIs into account, definite markers have to be included in the IRI pattern and other occurrences of this marker in the IRI prefix have to be escaped. This uniquely identifies Subtype map IRI prefixes and unambiguously distinguishes the table name and the column name from the rest of the IRI and from one another.

For an example that makes awkwardly – or fraudulently – chosen base IRIs introduce name clashes, see the paragraph about Entity map OWL class IRIs at the beginning of this section.

3.3.3. The proposed IRI generation scheme

This section introduces an IRI generation scheme meeting the requirements formulated in Section 3.3.1.

In this section, the following strings are subsumed under the term *marking strings*:
"`TABLE__`", "`TBL__`", "`PROP__`", "`REF__`" and "`SUBTYPE__`".
The string built by escaping (prefixing) all occurrences of marking strings or '~' characters in a string s with a '~' character will be called the *IRI-safe version* of s.

The IRI generation scheme is presented in Table 3.2. Here,
<base:> refers to the base IRI to be used for the generated IRI (see Section 2.1.1 – Basic concepts),
<cl. tbl name> refers to the table name of the database table concerning (see Section 3.3.2) in its IRI-safe version,
<cl. name 1st pk col> refers to the name of the first primary key column concerning (see Section 3.3.2) in its IRI-safe version,
</...> refers to the continuation of the previous pattern using the remaining primary key or foreign key columns,
<cl. col name> refers to the name of the column in question (see Section 3.3.2) in its IRI-safe version,
<cl. src tbl> refers to the table name of the database table containing the respective foreign key (see Section 3.3.2) in its IRI-safe version,
<cl. 1st src col> refers to the name of the first foreign key column of the respective foreign key (see Section 3.3.2) in its IRI-safe version,
<cl. tgt tbl> refers to the table name of the table referenced by the respective foreign key (see Section 3.3.2) in its IRI-safe version and
<cl. 1st tgt col> refers to the name of the first column referenced by the respective foreign key (see Section 3.3.2) in its IRI-safe version.

IRI type	Proposed IRI
Entity map OWL class IRI	<base:>`TABLE__`<cl. tbl name>
Identifier map IRI pattern	<base:>`TBL__`<cl. tbl name>`/`<cl. name 1st pk col>`/{$1}/</...>`
Attribute map OWL property IRI	<base:>`PROP__`<cl. tbl name>`__`<cl. col name>
Relation map OWL property IRI	<base:>`REF__`<cl. src tbl>`/`<cl. 1st src col>`</...>/`<cl. tgt tbl>`/`<cl. 1st tgt col>`</...>`
Subtype map IRI prefixes	<base:>`SUBTYPE__`<cl. tbl name>`__`<cl. col name>`/`

Table 3.2.: Proposed IRIs to be used in OBDA specification map fields

It is easily verified that the proposed IRI scheme is correct regarding the requirements described in Section 3.3.1: it provides unique IRIs for all types of IRIs it allows to create, regardless of the chosen base IRI (see proof in Section 3.3.4). Furthermore, the IRI scheme is expressive: ignoring the base IRI part, the kind of entity identified by the IRI can be determined by beginning of the IRI. In addition, it is regular in that the name of the containing table always occurs before the name of the first database column.

Taking the information in Section 3.3.2 into account, it is trivial to observe that the suggested IRI scheme, leaving out the demand of IRI-safe versions, is still correct, given that all IRIs are generated using the same base IRI.

Note that it is in any case necessary that the beginnings of all kinds of IRIs be mutually different: if, for example, an Identifier map IRI pattern also would commence with "<base:>`TABLE__`", a table named "`PERSONS/{17}`" – which is a valid table name for example in SQL [**sql**] – possibly could get an Entity map OWL class IRI assigned which clashes with the IRI resulting from the expansion of the IRI pattern "<base:>`TABLE__PERSONS/{$1}`".

3.3.4. Proof of correctness of the proposed IRI scheme

As described in Section 3.3.1, the previously described IRI schema is required to generate several types of IRIs without introducing name clashes, thus two equal IRIs for two distinct entities, independently of the chosen base IRIs. Additionally, the user shall be able to chose additional IRIs he can be sure won't collide with IRIs generated with the scheme from an infinite set.

In this proof, like in Section 3.3.2, the strings "`TABLE__`", "`TBL__`", "`PROP__`", "`REF__`" and "`SUBTYPE__`" are called *marking strings*.
Strings prefixed by '∼' are referred to as *escaped*, while strings not prefixed by '∼' are referred to as *unescaped*.

In the following, it is proven that the IRIs of each type do not clash, neither among themselves nor with IRIs of other types. Since all generated IRIs begin with a marking string, every IRI *not* beginning with a marking string, thus an infinite quantity, is sure not to collide with any of the generated IRIs, and so, the correctness regarding to the stated requirements is then proven.

Each Entity map OWL class IRI (including its base IRI) is of the form α`TABLE__`β, with α

not ending with '~' and α and β not containing any unescaped marking strings.
Thus, β is the table name, making the IRI unique among all other Entity map OWL class IRIs (see considerations in Section 3.3.2). Because the IRI does not contain any unescaped marking strings, it cannot collide with any IRI of another type and thus is indeed unique.

The proof for Identifier map IRI patterns, Attribute map OWL property IRIs, Relation map OWL property IRIs and Subtype map IRI prefixes is exactly analog.

□

4. The OBDA Specification Language (OSL)

TODO: aims, proceeding, structure

As described in [SGH+15], an OBDA specification consists of several types of maps, all containing data entries and links to other maps. This fits perfectly into the environment of ontologies and OWL, with data properties being the obvious choice to represent contained data entries and object properties being the obvious choice to represent links between maps. Also, a potential user probably to some degree is familiar with this environment, since this is what the bootstrapping process at the end amounts to.

Therefore, an ideal base for the OBDA Specification Language is OWL, being a solid framework for data and constraint representation with a high degree of software support, while imposing only a minimum of introductory preparation to the user.

Another advantage of this approach is that the specification is kept compact and focused on the entities that the language has to represent rather than primarily dealing with technical details. In particular, many of those details can be formulated as OWL restrictions in a header ontology demanded to be imported by documents conforming to the OSL specification. Thus, they are not only specified precisely but they are also stipulated in a machine-readable form for which tools are widely available, enabling the user to check many aspects of an OSL document for conformity with minimal effort.

4.1. Specification

[1] An OSL document is a valid OWL 2 document (as described in [W3C12]) containing individuals and data that represent the OBDA specification, as well as OWL properties that connect them. The individuals and OWL properties are recognized and mapped to their roles by their IRIs.

[2] An OSL document may contain more OWL entities (with IRIs not defined in this specification), which are ignored.

[3] An OSL document has to declare all individuals having different IRIs as different from each other (except those which are ignored, see Paragraph 2).
It is recommended to use the `owl:AllDifferent` OWL statement for this purpose.

[4] Unless stated otherwise, IRIs mentioned in the following are IRIs relative to a base IRI chosen by the user being empty (which makes the IRIs absolute [W3C09]) or ending with a hash character ('#').

Map type	OWL IRI
Entity map	<class URI>__ENTITY_MAP
Attribute map	<property URI>__ATTRIBUTE_MAP
Identifier map	<class URI>__IDENTIFIER_MAP
Relation map	<property URI>__RELATION_MAP
Subtype map	<class URI>__SUBTYPE_MAP
Translation table of attribute map	<property URI>__ATTRIBUTE_MAP__TRANSLATION_TABLE
Translation table of subtype map	<class URI>__SUBTYPE_MAP__TRANSLATION_TABLE

Table 4.1.: OWL individual IRIs in OSL

Map type	OWL class IRI
Entity map	`osl:EntityMap`
Attribute map	`osl:AttributeMap`
Identifier map	`osl:IdentifierMap`
Relation map	`osl:RelationMap`
Subtype map	`osl:SubtypeMap`
Translation table	`osl:TranslationTable`

Table 4.2.: OWL class membership of map representations in OSL

It is recommended to use that base IRI as `xml:base` XML attribute.
IRIs prefixed with `osl:` are IRIs relative to the IRI
http://w3studi.informatik.uni-stuttgart.de/~martispp/ont# .

[5] An OSL document has to import the following ontology (referred to as "the OSL header" in the following):
http://w3studi.informatik.uni-stuttgart.de/~martispp/ont/db2osl.owl

[6] The OWL individuals described by the OSL document representing the certain types of OBDA maps must have the IRIs specified in Table 4.1 (for base IRIs, see Paragraph 4). Here, *<class URI>* refers
to the `OWL class URI` field of the respective entity map for entity maps,
to the `OWL class URI` field of the associated entity map for identifier maps,
to the `OWL class URI` field of the associated entity map for subtype maps and
to the `OWL class URI` field of the entity map associated with the respective subtype map for translation tables of subtype maps.
Similarly, *<property URI>* refers
to the `OWL property URI` field of the respective attribute map for attribute maps (or, if it is empty, the value that would have been generated for it if it weren't empty),
to the `OWL property URI` field of the respective relation map for relation maps and
to the `OWL property URI` field of the respective attribute map for translation tables of attribute maps (or, if it is empty, the value that would have been generated for it if it weren't empty).

[7] The OWL individuals described by the OSL document representing the certain types of OBDA maps must be of the OWL types specified in Table 4.2 (for base IRIs, see Paragraph 4).

[8] The OWL properties described by the OSL document representing the fields of the certain OBDA maps must have the IRIs specified in Table 4.3 (for base IRIs, see Paragraph 4).

Map type	Field label	Field name	OWL IRI
Entity map	E1	Table name	osl:em__tableName
Entity map	E2	Label	osl:em__label
Entity map	E3	Identifier map	osl:em__identifierMap
Entity map	E4	Attribute maps...	osl:em__attributeMaps
Entity map	E5	OWL class URI	osl:em__owlClassURI
Entity map	E6	Description	osl:em__description
Attribute map	A1	Column name	osl:am__columnName
Attribute map	A2	SQL datatype	osl:am__sqlDatatype
Attribute map	A3	Mandatory	osl:am__mandatory
Attribute map	A4	Label	osl:am__label
Attribute map	A5	OWL property URI	osl:am__owlPropertyURI
Attribute map	A6	Property type	osl:am__propertyType
Attribute map	A7	Translation	osl:am__translation
Attribute map	A8	URI pattern	osl:am__uriPattern
Attribute map	A9	RDF language	osl:am__rdfLanguage
Attribute map	A10	XSD datatype	osl:am__xsdDatatype
Attribute map	A11	Description	osl:am__description
Identifier map	I1	Entity map	osl:im__entityMap
Identifier map	I2	Attribute maps...	osl:im__attributeMaps
Identifier map	I3	URI pattern	osl:im__uriPattern
Relation map	R1	Source entity map	osl:rm__sourceEntityMap
Relation map	R2	Source column	osl:rm__sourceColumns
Relation map	R3	Target entity map	osl:rm__targetEntityMap
Relation map	R4	Target column	osl:rm__targetColumns
Relation map	R5	OWL property URI	osl:rm__owlPropertyURI
Subtype map	S1	Entity Map	osl:sm__entityMap
Subtype map	S2	Column Name	osl:sm__columnName
Subtype map	S3	OWL superclass URI	osl:sm__owlSuperclassURI
Subtype map	S4	Prefix	osl:sm__prefix
Subtype map	S5	Suffix	osl:sm__suffix
Subtype map	S6	Translation	osl:sm__translation
Translation table	T1	Source value...	osl:tt__sourceValues
Translation table	T2	RDF ressource...	osl:tt__rdfRessources

Table 4.3.: OWL property IRIs in OSL

[9] The following OWL properties in the OSL document refer to lists of elements:

`osl:rm__sourceColumns`

`osl:rm__targetColumns`

`osl:tt__sourceValues`

`osl:tt__rdfRessources`

Therefore, they have the OWL class `osl:StringListNode` as their range, as is required by the OSL header. They must connect the respective individual to an `osl:StringListNode` individual in every case. This "root node" must *not* have an `osl:hasValue` property.
If the represented list is not empty, the list elements are represented by other `osl:StringListNode` individuals connected seriatim by the property `osl:nextNode`, with the first individual being connected to the root node. The node representing the last list element must not have an `osl:nextNode` property.
All nodes except the root node *may* have an `osl:hasValue` property connecting them to their values. The actual list consists of exactly these values, thus, nodes without values are ignored.
It is recommended to enumerate the node IRIs, using 0 for the root node.

5. The db2osl software

TODO: uris

Besides the conception of the OBDA Specification Language (OSL), the design and implementation of the db2osl software was an important part of this work. The program itself and its creation process are described in the following sections: Section 5.1 describes the functionality the program offers. Section 5.2 describes how this functionality is exposed to the program environment. Section 5.3 describes the program architecture both on a coarse and a fine level. Section 5.4 mentions some numbers and statistics about the program. Implementation topics are dealt with in Chapter 6. For detailed descriptions of the classes and packages of db2osl, refer to Appendices TODO.

This chapters' sections present the information in a functionally-structured fashion: the concepts and decisions are described along with the topics they are linked to and the problems that made them arise.

5.1. Functionality

As described in the introduction of this thesis, the db2osl software is a program automatically deriving an OBDA specification from a relational database schema, which then can be used by other tools to drive the actual bootstrapping process. Its functionality is described in this section, leaving out self-evident features, and is then listed completely. How this functionality is exposed to users is described in Section 5.2 – Interface and usage. The bootstrapping process using direct mapping as the core functionality of the software is described in Section 3.1.

TODO: reference to OBDA topics

The database schema is retrieved by connecting to an SQL database and querying its schema information. Parsing SQL scripts or SQL dumps currently is not supported. The databases to derive information from can be specified by regular expressions, while there are also options to use other databases than specified or even other database servers, taken from a list of hard-coded strings. While these features may not seem to carry real benefit at the first glance, they proved to be useful for testing purposes, especially since the retrieval of a database schema can take some time TODO (see Section 5.4 – Numbers and statistics). For the same purpose, db2osl allows the processing of a hard-coded example database schema.

In addition to OSL output, a low-level output format containing information on all fields of the underlying objects is supported, which is useful for debugging (however, this feature has to be enabled via one slight change in the source code). To allow for some customization, the insertion of an own OSL header is supported (for more information on the OSL header, see the specification of the OBDA Specification Language in Section 4.1). If the standard OSL header is used, it is by default loaded from a hard-coded copy, so bootstrapping information

from a database server running locally or from the hard-coded example schema requires no Internet connection (simply inserting the `owl:imports` statement of course would not anyway, but the generated underlying ontology is always checked for consistency with the OSL header to prevent the generation of invalid output).

The DB2OSL software can be used both in an interactive and in a non-interactive mode, while skipping a database or a database server or aborting the entire bootstrapping process is possible in either mode. Multiple database servers can be specified for a bootstrapping operation, which then are checked in order for a matching database, allowing to make use of mirrors or fallback servers. Additionally, multiple bootstrapping operations can be specified to be performed in sequence with one invocation of DB2OSL, while all features and settings previously described are enabled, disabled or set per operation. Finally, a help text can be displayed which describes the usage of DB2OSL including the description of all command-line arguments.

The functionality of the DB2OSL software can be summarized as follows:

- Bootstrap one or more OBDA specifications from a database schema by connecting to an SQL database server
- Specify a custom port, login and password for the database server
- Ask for passwords interactively (before starting any bootstrapping operation), hide them if desired
- Specify database names by regular expressions
- Process an arbitrary database if the specified database could not be found or unconditionally
- Connect to a database server containing example databases without having to specify any further details
- Process a hard-coded example database schema without having to specify any further details
- Use the OSL format described in Chapter 4 – The OBDA Specification Language (OSL) or a detailed low-level format for output (the latter is for debugging purposes and has to be enabled in the source code)
- Write to standard output or to a file
- Insert a custom OSL header (see the specification of the OBDA Specification Language (OSL) in Section 4.1 for details)
- Consistency check against a custom OSL header
- Consistency check against the standard OSL header without internet connection
- Act interactively or non-interactively
- Skip currently retrieved database (and try next on server), skip current server or abort the overall process at any time, even in non-interactive mode
- Define multiple database servers to check in order for the specified database
- Specify multiple bootstrapping operations to perform in order

- Configure the features described in the above notes per bootstrapping operation
- Display a help text describing the usage of DB2OSL, including the description of all command-line arguments

5.2. Interface and usage

This section describes the interface to the operating system and the user interface. For information on programming interfaces, see Section 5.3 – Architecture of DB2OSL.

5.2.1. User interaction and configuration

Basic usage

Currently, the only user interface of DB2OSL is a command-line interface. Since the program is supposed to bootstrap the OBDA specification automatically and thus there is little interaction, but a lot of output, this was considered ideal. Basically, one invocation of DB2OSL will initiate the automatic, non-interactive bootstrapping of exactly one OSL specification written to the standard output, a behavior which can be modified via command-line arguments. Because of its ability to write to the standard output (which is also the default behavior), it is easy to pipe the output of DB2OSL directly into a program that handles it in a Unix-/POSIX-like fashion [McI87]:

```
db2osl myserver.org | osl2onto myserver.org
```

(supposed OSL2ONTO is a tool that reads an OSL specification from its standard input and uses it to bootstrap an ontology from the database specified on its command line).
This scheme is known as "Pipes and Filters architectural pattern" [BMRSS96].

By inserting additional "filters", the bootstrapping process can be customized without changing any of the involved programs:

```
db2osl mydatabase.org | customize_spec.sh | osl2onto mydatabase.org
```

(supposed CUSTOMIZE_SPEC.SH is a shell script that modifies a given OSL specification in the way the user desires).

Configuration via command-line arguments

The behavior of DB2OSL itself can be adjusted via command-line arguments (only). Most features can be configured via short options (as, for example, `-P`). To allow for enhanced readability of DB2OSL invocations, each feature can (also) be configured via a long option (like `--password`). The utilization of configuration files was considered, but for the time being seen as unnecessary complicating while not addressing any real difficulties.

The command-line arguments DB2OSL currently supports are described in Table 5.1; their default values are listed in Table 5.2. There is currently no switch to set the output format,

Option(s)	Description, taken from the help page of DB2OSL
`--database`, `-d`	database name (JAVA regular expression) databases have to match to be processed; see also: `--loose-database-match`
`--echo-password`	echo input when prompting for SQL password – must be specified before `--password-prompt` to get effective
`--help`, `-h`,	show this help and exit
`--interactive`, `-i`	be interactive when chosing database
`--login`, `-L`	SQL login
`--loose-database-match`	if no database matching the regex specified with `--database` is found on the given server and `--interactive` is not specified for this job, use some other database
`--osl-header`	use the specified custom (non-standard) OSL header, implies `--remote-osl-header` (to import no header, specify the empty string)
`--output-file`, `-o`	use the specified output file (for the standard output, specify "-")
`--password`, `-P`	SQL password; use `--password-prompt` to get a password prompt (if you do both, the password set via this switch will be ignored)
`--password-prompt`, `-p`	prompt for SQL password; a password set via `--password` is ignored
`--remote-osl-header`, `-R`	don't use hard-coded version of the OSL header for verification
`--remote-test`	try to retrieve a database schema from a hard-coded list of servers and take the first one successfully retrieved (and accepted, when `--interactive` is given; note: give a dummy server if you want to do a test besides other jobs)
`--test`	use hard-coded test database schema, ignore given servers (note: give a dummy server if you want to do a test besides other jobs)

Table 5.1.: Command-line arguments in DB2OSL – descriptions

since the only supported output format, besides OSL, is a low-level output format for debugging purposes. Because of this and since the change that has to be made in the source code to enable it only involves changing one token, it was preferred not to offer a command-line option for this, to not unnecessarily complicating the command-line interface for the normal, non-debugging, user.

The sole invocation of `db2osl`, without any arguments, does not initiate any processing but displays the usage directions instead, in addition to an error message pointing out the missing server argument.

Multiple bootstrapping operations or multiple servers

To perform multiple bootstrapping operations with only one invocation of DB2OSL, it is sufficient to concatenate the command-line arguments for each operation, separated by blanks, to get the final command line. However, when combining a test job with other operations, some

Option(s)	Default value
--database, -d	.*
--echo-password	false
--help, -h,	false
--interactive, -i	false
--login, -L	anonymous
--loose-database-match	false
--osl-header	<empty string>
--output-file, -o	-
--password, -P	<empty string>
--password-prompt, -p	false
--remote-osl-header, -R	false
--remote-test	false
--test	false

Table 5.2.: Command-line arguments in DB2OSL – default values

arbitrary string has to be inserted as dummy server to allow distinguishing the different jobs and assigning each command-line argument to the appropriate job.

Likewise, to check several servers in order for the database to be used for one bootstrapping operation, these servers have to be concatenated, separated by blanks. Again, the distinction of the different bootstrapping jobs has to be possible, so all but the first operation have to have at least one command-line argument that signals the beginning of a new job definition (which is no practical problem, since to enforce this, a default argument simply can be stated explicitly without changing the behavior of the invocation).

All settings are configured per operation, so, when using a shell that separates batched commands by ';',

```
db2osl --database employees --password itsme sql.myemployer.com
--database test myserver.org backup.myserver.org
```

is equivalent to

```
db2osl --database employees --password itsme sql.myemployer.com;
db2osl --database test myserver.org backup.myserver.org
```

Thus, a parameter defined for one operation (like the password in the example) will have no effect on other operations. This ensures that typical errors are prevented when merging several invocations of DB2OSL into one (or vice versa) and allows for a straight-forward and comprehensible implementation.

Advanced modifications

Since OSL specifications are plain text files, a user can edit them in any desired text editor if he wants to change them in ways that go beyond the functionality DB2OSL provides or that can be achieved by scripts or programs modifying their input automatically. Because of OSL

being a subset of OWL (see the specification of OSL in Section 4.1), he can thereby take advantage of editors supporting syntax highlighting or other features making the handling of the respective OWL serialization more comfortable.

Moreover, every common ontology editor can be used to edit the generated OSL specification automatically or manually. Doing so, care has to be taken to make the ontology remain a conforming OSL specification. However, since the restrictions imposed by OSL are rather small and intuitive, this is easily achieved. One of the most popular ontology editors [MBSF04], Protégé, is an open, Java based platform supporting plug-ins [NCFK+03]; for a habitual Protégé user it should be an easy task to write an OSL plugin. Furthermore, upcoming tools supporting OSL (see Section 7.2 – Future work) most likely will be able to check input files for conformity with the OSL definition.

5.2.2. Integration into systems

Besides the use cases described in Paragraph "Basic usage" and in Paragraph "Advanced modifications" in the previous Section 5.2.1, there are many other ways in which db2osl can be used. For example, a database can be periodically checked for changes that make a re-bootstrapping necessary:

```
db2osl -d mydb myserver.org | sha256sum >oldsum
cp oldsum newsum
while diff oldsum newsum; do  # while checksums are the same
      sleep 3600  # wait 1 hour
      db2osl -d mydb myserver.org | sha256sum >newsum
done
rm oldsum newsum
# notify web admin via e-mail:
date | mutt -s "Re-bootstrapping necessary" web-admin@myserver.org
```

Another possible example is the integration of db2osl into a shell script that bootstraps all databases on a server:

```
regex='(?!$).*'  # accept all nonempty database names first
while db2osl -d "$regex" -o spec myserver.org; do
      dbname="` sed -ne '/xmlns:ont/ { s|.*/||; s|#"||p }' spec `"
      mv spec "$dbname".osl
      # don't use this database a second time:
      regex="` printf %s "$regex" | sed -e "s,\\\\$,$|$dbname$," `"
done
```

Since the programming language used to implement db2osl is Java, it is possible to deploy it on all platforms offering the Java Runtime Environment TODO. Additionally, it is possible to deploy it as a Web application TODO.

To simplify integration on the code level, the architecture of db2osl was designed to be highly modular and to cleanly separate code with different areas of responsibility into different packages (for details about the structuring of db2osl, see Section 5.3 – Architecture of db2osl).

This modularity, besides facilitating understanding the code, allows for a high degree of code reusability.

For example, the packages `database`, `osl` and `specification` can be reused in other programs with little or no changes – the biggest change involves combining the database schema retrieval with the user interface of the new program to provide control over the retrieval process when reusing the `database` package (to do this, three method calls have to be replaced). If the accruing information shall be used in another way than being output or logged, this of course has to be implemented. If not, it is sufficient to replace the used `Logger` object by anohter one providing the desired behavior, since the `Logger` class is part of the JAVA API and widely used [Gup03]. This is a good example of how using well-known and commonly used classes can greatly improve modularity and reusability.

5.3. Architecture of db2osl

5.3.1. Libraries used in db2osl

5.3.2. Coarse structuring of db2osl

TODO: overall description, modularity, extendability, ex: easy to add new in-/output formats
TODO: mapping profiles (maybe better in next subsection) TODO: Java, OPTIQUE

Package structuring of db2osl

The 45 classes of DB2OSL were assigned to 11 packages, each containing classes responsible for the same area of operation or taking over similar roles. Care was taken that package division happened senseful, producing meaningful packages with obvious task fields on the one hand, while on the other hand implementing an incisive separation with a notable degree of decoupling. Packages were chosen not to be nested but to be set out vapidly. Since this doesn't have any functional implications [Sch14], but is rather an implementation detail, this is further explained in Section 6.2.5 – Use of packages.

The packages are introduced and described in Table 5.3. The lists of classes each package contains are given in Table A.1 in Appendix A.1 – Package contents (DB2OSL).

Package	Description
`bootstrapping`	Classes performing bootstrapping
`cli`	Classes related to the command line interface of DB2OSL
`database`	Classes related to the representation of relational databases and attached tasks
`helpers`	Helper classes used program-wide
`log`	Classes related to logging and diagnostic output
`main`	The `Main` class
`osl`	Classes representing OBDA specifications (as described in [SGH+15]) using the OBDA SPECIFICATION LANGUAGE (OSL)
`output`	Classes used to output OBDA specifications as described in [SGH+15]
`settings`	Classes related to program and job settings (including command line parsing)
`specification`	Classes representing (parts of) OBDA specifications (as described in [SGH+15]) directly, without involving OSL
`test`	Classes offering testing facilities

Table 5.3.: Descriptions of the packages in DB2OSL

Besides intuition, as stated, care was involved when partitioning the program into these packages, which included the analysis of the package interaction under a given structure, and the carrying out of changes to make this structure achieve the desired pronounced decoupling with limited and intelligible dependencies.

The `main` package was introduced to make the `Main` class, which carries information needed by other packages – most prominently, the program name –, `import`able from inside these packages. For this, it is required for `Main` not to reside in the `default` package [Sch14].

Decoupling some of the functionality of a package into a new package – which, in a nesting package structure, most probably would have become a sub-package – and thus sacrificing the benefit of having fewer packages also played a role in some cases. Namely, `osl` is a package on its own instead of being part of the `specification` package, the `bootstrapping` classes also form a package on their own instead of belonging to the `specification` package, the classes of the `log` and the `cli` packages were not merged into one package, although logging currently exclusively happens on the command line, and the functionality of the `test` package, though containing only a few lines of code, was separated into its own package.

Even though the package structure would have become quite simpler with these changes applied – 4 out of 11 packages could have been saved this way – the first aim mentioned – meaningfulness and intuitiveness – was taken seriously and the presented partitioning was considered a more natural and comprehensible structuring, emphasizing different roles and thus being a more proper foundation for future extensions of the program. For example, because the `bootstrapping` package is central to the program and takes over an active, processing role and in that is completely different from the classes of the `specification` package which on their part have a *representing* role, it was considered senseful not to merge these two packages. This undergirds the separation of concerns within the program and stresses that the functionality of the `bootstrapping` package should not interweave with that in the `specification` package, making it easier for both to stay independent and further develop into understandable and suitable units.

Package interaction

As mentioned, the structuring of the packages was driven by the aim to gain a notable amount of decoupling. How this reflected in the dependency structure, thus the classes from other packages that the classes of a package depend on, is described in the following. As was also mentioned, the information presented here also acted back on the package partitioning, which changed in consequence.

Dependencies of or on package `helpers` are not considered in the following, since this package precisely was meant to offer services used by many other packages. In fact, all facilities provided by `helpers` could just as well be part of the Java API, but unfortunately are not. The current dependency structure, factoring in this restriction, is shown in Figure 5.1 and reveals a conceivably tidy system of dependencies.

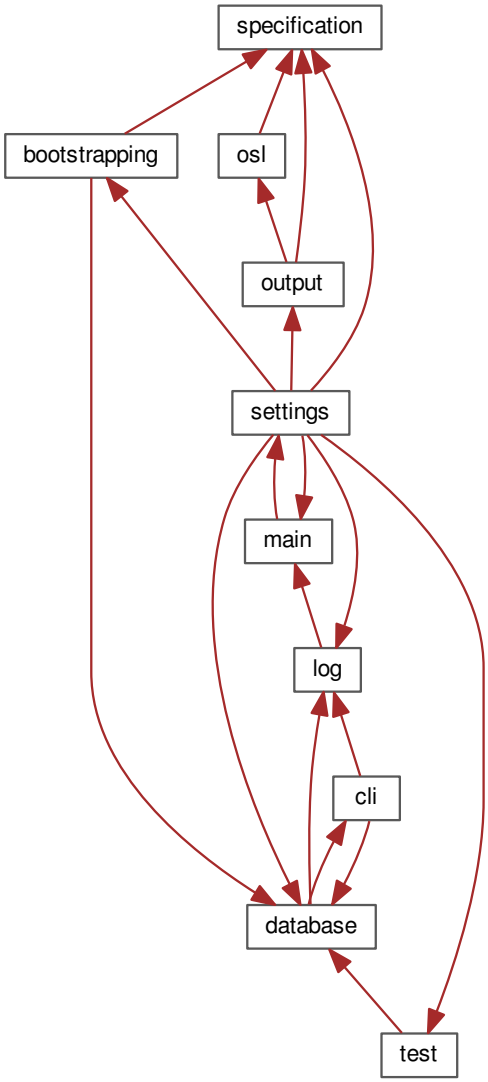


Figure 5.1.: Package dependencies in DB2OSL. "→" means "depends on".

Except for the package `settings` (which is further explained below), every package has at most two outgoing edges, that is packages it depends on. Previous versions of DB2OSL had a quite more complicated package dependency structure, depicted in Figure 5.2. In this previous package structure, the maximum number of dependencies of packages other than `settings` on other packages is three, which also seems reasonably less. However, in the new structuring, `specification` has no packages it depends on and thus suits its purpose of providing a mundane and straight-forward representation of an OBDA specification much better.

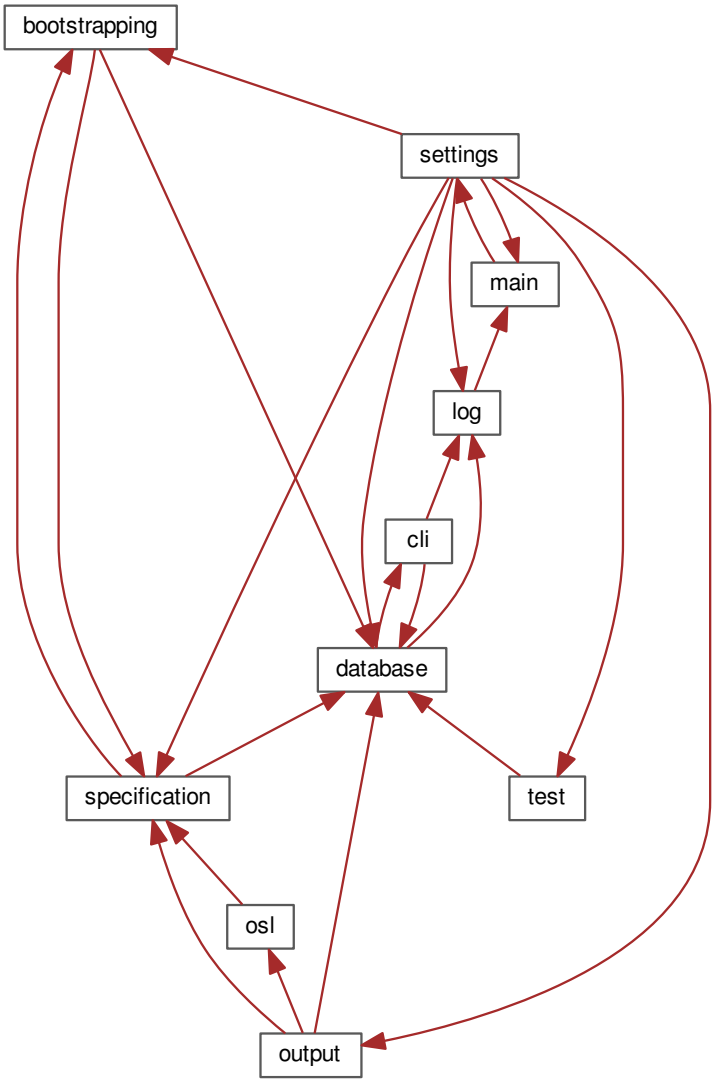


Figure 5.2.: Package dependencies in earlier versions of DB2OSL. "→" again means "depends on".

Though there still are quite a number of dependencies (to be precise: 19), many of them (8, thus, nearly half) trace back to one central package in the middle, `settings`. This may seem odd at first glance, considering that most of the edges connecting to the `settings` node are outgoing edges and only one is incoming, whereas in a design where the settings are configured from within a single package and accessed from many other packages this would be the other way round. The reason for this constellation is that, as described in Section 5.2 – Interface and usage, all settings in DB2OSL are configured per bootstrapping job (there are no global settings) and so `settings` contains a class `Job` (and currently no other classes), which represents the configuration of a bootstrapping job but also provides a `perform()` method combining the facilities offered by the other packages.

By this means, the `perform()` method of the `Job` class acts as the central driver performing

the bootstrapping process, reducing the `main()` method to only 7 lines of code and turning `settings` into something like an externalized part of the `main` package. If, in a future version of the program, this approach is changed and global settings or configuration files are introduced, `settings` will still be the central package, leaving the package structure and dependencies unchanged, since it either way contains information used by many other packages. This was the reason why it was not renamed to, for example, `driver`, which was considered, since at first glance it seems quite a bit unnatural to have the driver class reside in a package called "settings".

5.3.3. Fine structuring of db2osl

TODO: OBDA spec rep

While the packages in DB2OSL are introduced and described in Section 5.3.2 – Coarse structuring of DB2OSL, the classes that comprise them are addressed in this section. For a list of classes contained in each package, refer to Appendix A.1.

TODO: total classes etc.

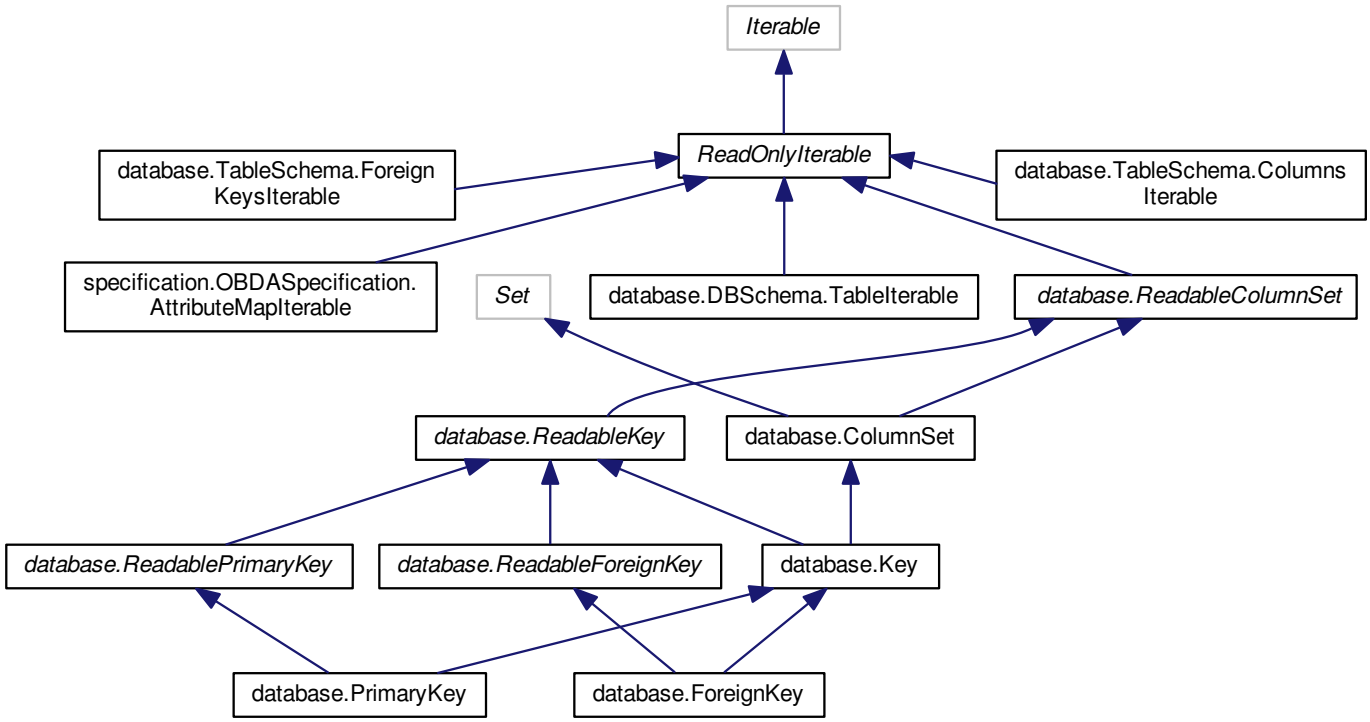
Package contents

Table A.1 lists the classes each package contains. The packages `cli`, `main`, `osl` and `settings` contain only one class each, while the by far most extensive package is `database`, containing 15 classes.

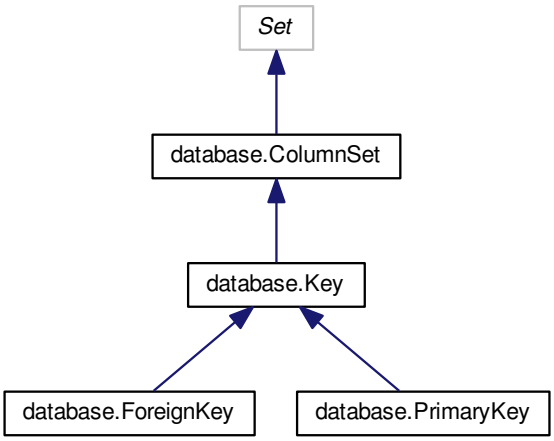
Class organization

Organizing classes in a structured, obvious manner such that classes have well-defined roles, behave in an intuitive way, ideally representing artifacts from the world modeled in the program directly [Str00], is a prerequisite to make the code clear and comprehensible on the architectural level.

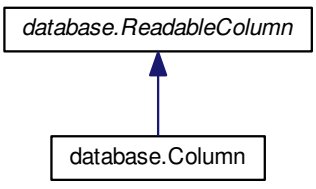
Section 6.2.4 – Use of classes as part of Section 6.2 – Code style describes the identification and naming scheme for the classes in DB2OSL. However, it is also important, to arrange these classes in useful, comprehensible class hierarchies to avoid code duplication, make appropriate use of the type system, ease the design of precise and flexible interfaces and enhance the adaptability and extensibility of the program.



(a) `ColumnSet` class hierarchy in DB2OSL

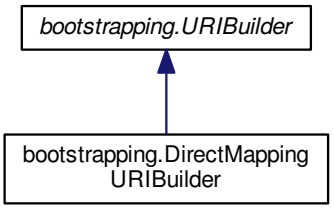


(b) `ColumnSet` class hierarchy in DB2OSL – simplified

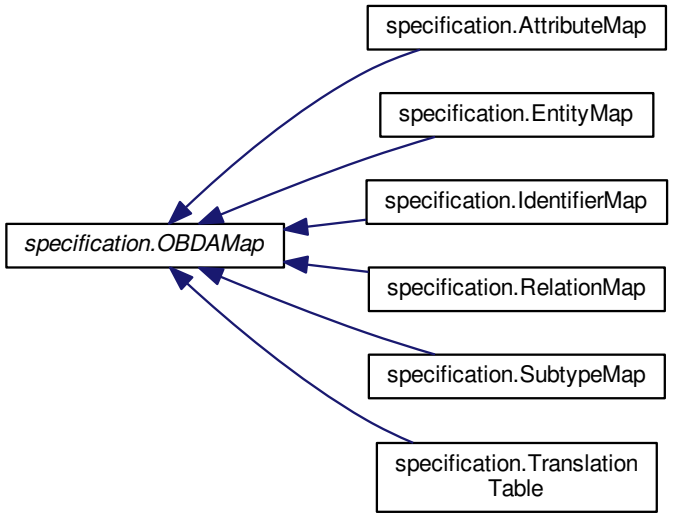


(c) `Column` class hierarchy in DB2OSL

Figure 5.3.: Database class hierarchies in DB2OSL. Interface names are italicized, external classes or interfaces are hemmed with a gray frame.

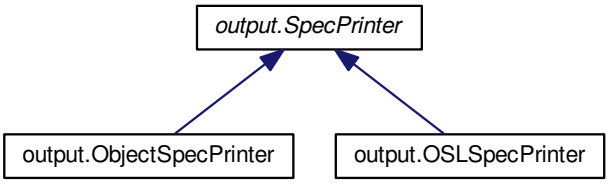


(a) `URIBuilder` class hierarchy in DB2OSL

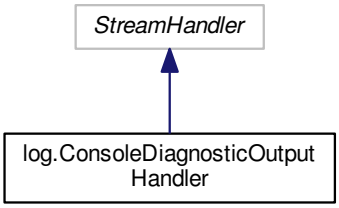


(b) `OBDAMap` class hierarchy in DB2OSL

Figure 5.4.: OBDA specification class hierarchies in DB2OSL. Interface names are italicized, external classes or interfaces are hemmed with a gray frame.



(a) `SpecPrinter` class hierarchy in DB2OSL



(b) `StreamHandler` class hierarchy in DB2OSL

Figure 5.5.: Logging and output class hierarchies in DB2OSL. Interface names are italicized, external classes or interfaces are hemmed with a gray frame.

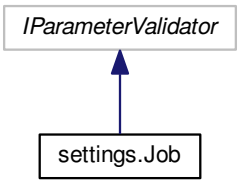
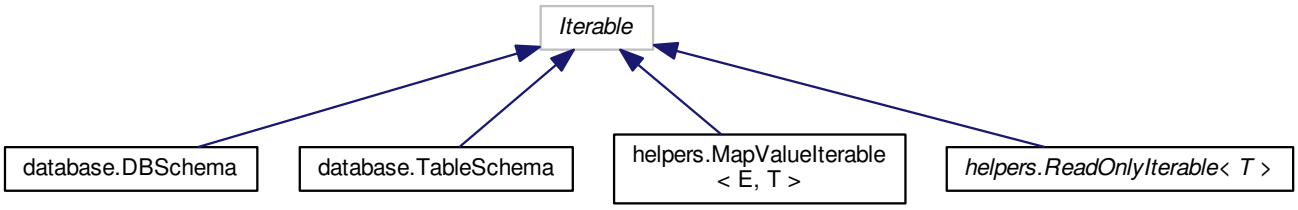
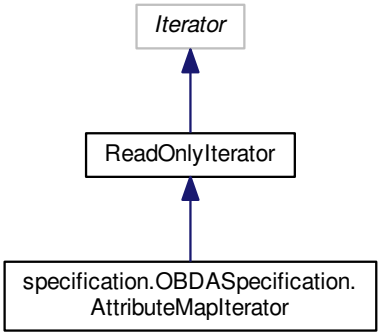


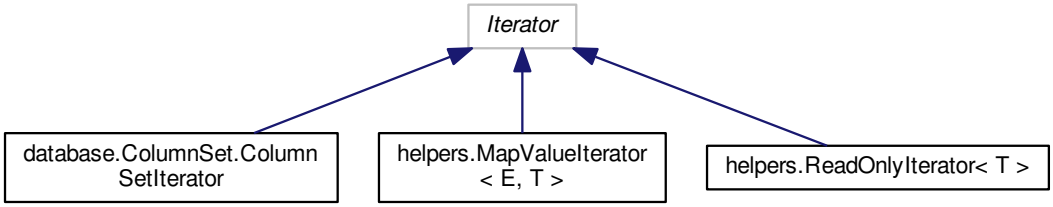
Figure 5.6.: `Job` class hierarchy in DB2OSL. Interface names are italicized, external classes or interfaces are hemmed with a gray frame.



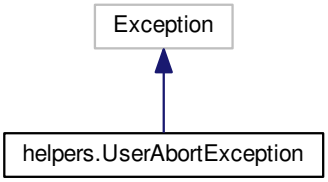
(a) `Iterable` class hierarchy in DB2OSL



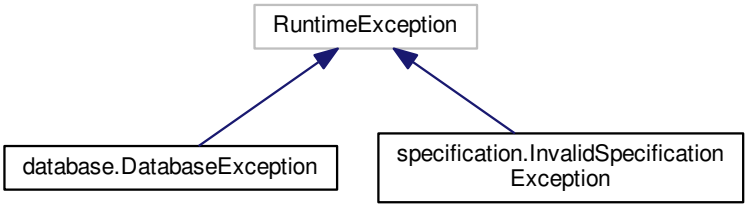
(b) `ReadOnlyIterator` class hierarchy in DB2OSL



(c) `Iterator` class hierarchy in DB2OSL



(d) `Exception` class hierarchy in DB2OSL



(e) `RuntimeException` class hierarchy in DB2OSL

Figure 5.7.: Miscellaneous class hierarchies in DB2OSL. Interface names are italicized, external classes or interfaces are hemmed with a gray frame.

- `main.Main`
- `database.RetrieveDBSchema`
- `database.Table`
- `helpers.Helpers`
- `helpers.SQLType`
- `specification.OBDASpecification`
- `osl.OSLSpecification`
- `bootstrapping.Bootstrapping`
- `cli.CLIDatabaseInteraction`
- `log.GlobalLogger`
- `test.CreateTestDBSchema`
- `test.GetSomeDBSchema`

Table 5.4.: Standalone classes in DB2OSL

Note that every class hierarchy has at least one `interface` at its top. Classes not belonging to a class hierarchy were chosen not to be given an interface "factitiously", which would have made them part of a (small) class hierarchy [Sch14]. Deliberately, the scheme often recommended [GHJV95] to give every class an interface it `implements` was not followed but the approach described by Stroustrup [Str13] to provide a rich set of so called "concrete types" not designed for use within class hierarchies, which "build the foundation of every well-designed program" [Str13]. The details of this consideration are explained in Paragraph "Java interfaces" in Section 6.2.4 – Use of classes. In fact, many useful types were already offered by the JAVA API and of course were not re-implemented.

Class `Column` with its interface `ReadableColumn` is an exception in that it was given an interface although it is basically a concrete type. The reason for this is the chosen way to implement const correctness, described in Paragraph "Const correctness" (which is part of Section 6.2.4 – Use of classes) TODO. This technique forced class `Column` to `implement` an interface, thus needlessly making it part of a class hierarchy, but also complicated the structure of some class hierarchies. Consider the class hierarchy around `ColumnSet`, shown in Figure 5.3b. Definitely, it seems overly complicated at the first glance. But this complexity solely is introduced by the artificial `Readable...` interfaces; would JAVA provide a mechanism like C++'s `const`, this hierarchy would be as simple as in TODO.

However, since const correctness is an important mechanism effectively preventing errors while on the other hand introducing clarity by itself, it was considered too important to be sacrificed, even for a cleaner and more intuitive class hierarchy. The fact that the `Readable...` scheme is very straight-forward and a programmer reading the documentation knows about its purpose and the real, much smaller, complexity also makes some amends for the simplicity sacrificed. The const correctness mechanism itself thereby hinders uninformed or ignorant programmers from mistakenly using the wrong class in an interface in many cases.

For more information about the program structure on the class level, see Section 6.2 – Code style, while for a detailed class index refer to Appendix TODO.

5.4. Numbers and statistics

TODO: consequences The following numbers and statistics can be stated about DB2OSL:

Subject	Value	Details in sections
Number of classes/interfaces	45	5.3.3, 6.2.4, 6.2.4
Number of packages	11	5.3.2, 6.2.5
Classes per package	ca 4.1	5.3.2
Number of methods	TODO	
Number of comments		6.2.1
Number of JAVADOC comments		6.2.1, 6.1 (TODO)
Number of non-JAVADOC comments		6.2.1
Average length of non-JAVADOC comments		6.2.1
Lines of code (LOC)	TODO	
Non-comment lines of code (NCLOC)	TODO	
Average NCLOC length	TODO	
NCLOC per comment	TODO	6.2.1
NCLOC per function	TODO	
NCLOC per class	TODO	
NCLOC per package	TODO	
Average NCLOC per method	TODO	6.2.1
TODO: Method/class coupling	TODO	5.3.3
Deepest nesting level	5	6.2.2
Retrieval time for example schema	TODO	5.4.1
Bootstrapping time for example schema	TODO	5.4.1

Table 5.5.: Numbers and statistics about DB2OSL

5.4.1. Benchmarking details

The example schema

TODO: description, URI

The benchmark process

TODO: description

The benchmark system

TODO: description

6. Implementation of db2osl

TODO: intro

Section 6.1 explains what tools where used to create the program. Section 6.2 describes concepts and decisions that where implemented on the code level to yield clean code.

6.1. Tools employed

Several tools were used for the creation of DB2OSL, some of which also proved useful during the creation of this thesis. Their use is described briefly in this section.

Thank is proffered to the contributers of these tools, all of which are free and open-source software TODO.

Debian GNU/Linux

The operating system to run the other tools on was DEBIAN GNU/LINUX, version 8.0 ("Jessie").

Basic Unix tools

Some of the basic Unix shell tools, namely FIND, CAT, GREP, SED, LESS, DIFF and, of course, the shell itself (BASH was used) were very useful, for instance, for searching all source code files for common errors or for remains of obsolete constructs that were replaced, for carrying out changes on all source files and for detecting and removing debugging code.

All of the tools were implementations created as part of the GNU project.

git

GIT was used both for version control and for shared access to the source code and related artifacts.

vim

To apply changes involving advanced regular expressions, to perform block editing (insert or remove columns from multiple lines at once), to insert debugging code and similar editing tasks, VIM was very useful.

Eclipse

The IDE to develop the program in was Eclipse. It proved very useful particularly due to its abilities to easily create packages and move source files between them, to ease the creation of in-code documentation and other useful features like automatic indention or the automatic insertion of `final` keywords.

OpenJDK

The Java compiler, the Javadoc tool (see next paragraph), the Java debugger and the Java Runtime Environment used were the implementations provided by the OpenJDK project, version 7.

Javadoc

Javadoc was used as the primary documentation generation system due to its ability to create clear and well-arranged documentations. Besides, documentations created by Javadoc are familiar to most Java programmers and cleanly integrate into the Java environment; for example, methods (automatically) inherited from the `Object` class are incorporated and links to methods and classes provided by the Java API are automatically generated.

Doxygen

To complement the documentation generated by Javadoc (see previous paragraph), Doxygen was used, which supports all used Javadoc constructs. This was particularly sensible, because Doxygen is able to create a much more in-depth documentation, that for instance includes `private` and `protected` members, the complete source code with syntax highlighting and references to it and detailed dependency and call graphs for all classes or methods, respectively. Thus, the Doxygen documentation is meant to be a more extensive, detail-oriented documentation providing insight into implementation issues.

6.2. Code style

TODO: Conventions, ex.: iterators
As the final system hopefully will have a long lifetime cycle and will be used and refined by many people, high code quality was an important aim. Beyond architectural issues this also involves cleanness on the lower level, like the design of classes and the implementation of methods. Common software development principles were followed and the unfamiliar reader was constantly taken into account to yield clean, readable and extensible code.

6.2.1. Comments

Comments were used at places ambiguities or misinterpretations could arise, yet care was taken to face such problems at their roots and solve them wherever possible instead of just effacing the ambiguity with comments. This approach is further explained in Section 6.2.2 – "Speaking code" and rendered many uses of comments unnecessary.

In fact, the number of (plain, e.g. non-JAVADOC) comments was consciously minimized, to enforce speaking code and avoid redundancy. An exception from this was the highlighting of subdivisions. In class and method implementations, comments like

```
//********************** Constructors **********************\\
```

were deliberately used to ease navigation inside source files, but also to enhance readability: parts of method implementations, for example, were optically separated this way. Another alternative would have been to use separate methods for these code pieces, and thereby sticking strictly to the so-called "Composed Method Pattern" [Bec97], as was done in other cases. However, sticking to this pattern too rigidly would have introduced additional artifacts with either long or non-speaking names, would have interrupted the reading flow and also would have increased complexity, because these methods would have been callable at least from everywhere in the source file. Consequently, having longer methods at some places that are optically separated into smaller units that are in fact independent from each other was considered an elegant solution, although, surprisingly, this technique does not seem to be proposed that often in the literature.

Wherever possible, the appropriate JAVADOC comments were used in favor of plain comments, for example to specify parameters, return types, exceptions and links to other parts of the documentation. This proved even more useful due to the fact that DOXYGEN supports all of the used JAVADOC comments [Hee16] (but not vice versa [Ora16]).

6.2.2. "Speaking code"

As mentioned in Section 6.2.1 – Comments, the code was tried to be designed to "speak for itself" as much as possible instead of making its readers depend on comments that provide an understanding. In doing so, besides reducing code size due to the missing comments, clean code amenable to unfamiliar readers and unpredictable changes was enforced. This is especially important since, as described in Section 5.3 – Architecture of DB2OSL, DB2OSL was designed to not only be a standalone program but also offer components suitable for reusability.

TODO: understandability <- code size

The following topics were identified to be addressed to get what can be conceived as "speaking code":

- Meaningful typing
- Method names
- Variable names
- Intuitive control flow

- Limited nesting
- Usage of well-known structures

The rest of this section describes these topics in some detail. Besides, an intuitive architecture and suitable, well-designed libraries also contributed to the clarity of the code (TODO: move).

Meaningful typing

Meaningful typing includes the direct mapping of entities of the modeled world to code entities [Str13] as well as an expressive naming scheme for the obtained types. Furthermore, inheritance should be used to express commonalities, to avoid code duplication and to separate implementations from interfaces [Str13].

All real-world artifacts to be modeled like database schemata, tables, table schemata. columns, keys and OBDA specifications with their certain map types were directly translated into classes having simple predicting names like `Table`, `TableSchema` and `Key`. Package affiliation provided the correct context to unambiguously understand these names.

Method names

Assigning expressive names to methods is a substantially important part of producing speaking code, since methods encapsulate operation and as such are important "building blocks" for other methods [Str13] and ultimately the whole program. Furthermore, method names often occur in interfaces and therefore are not limited to a local scope, and neither are easily changeable without affecting callers [Sch14].

Ultimately, care was taken that method names reflect all important aspects of the respective method's behavior. Consider the following method from CLIDatabaseInteraction.java:

```
public static void promptAbortRetrieveDBSchemaAndWait
     (final FutureTask<DBSchema> retriever) throws SQLException
```

It could have been called `promptAbortRetrieveDBSchema` only, with the waiting mentioned in a comment. However, the waiting (blocking) is such an important part of its behavior, that this was considered not enough, so the waiting was included in the function name. Since the method is called at one place only, the lengthening of the method name by 7 characters or about 26 % is really not a problem.

Variable names

To keep implementation code readable, care was taken to name variables meaningful yet concise. If this was not possible, expressiveness was preferred over conciseness.

For example, in the implementation of the database schema retrieval, variables containing data directly obtained from querying the database and thus being subject to further processing was consequently prefixed with "`recvd`", although in most cases this technically would not have been necessary.

Intuitive control flow

To consequently stick to the maxim of speaking code and further increase readability, control flow was tried to kept intuitive. `do-while` loops, for example, are unintuitive: they complicate matters due to the additional, unconditional, loop their reader has to keep in mind. Even worse, JAVA's Syntax delays the occurrence of their most important control statement – the loop condition – till after the loop body. Usually, `do-while` loops can be circumvented by properly setting variables influencing the loop condition immediately before the loop and using a `while` loop. Consequently, `do-while` loops were omitted – the code of DB2OSL does not contain a single `do-while` loop. TODO: references

Another counterproductive technique is the avoidance of the advanced loop control statements `break`, `continue` and `return` and the sole direction of a loop's control flow with its loop condition, often drawing on additional boolean variables like `loopDone` or `loopContinued`. This approach is an essential part of the "structured programming (paradigm)" [Dij72] and its purpose is to enforce that a loop is always left regularly, by unsuccessfully checking the loop condition, which shall ease code verification [Dij72]. A related topic is the general avoidance of the `return` statement (except at the end of a method) for similar considerations [Dij72]. However, both are not needed [Mar08] and, as always, the introduction of artificial technical constructs impairs readability and the ability of the code to "speak for itself".

Consequently, control flow was not distorted for technical considerations and care was taken to yield straight-forward loops, utilizing advanced control statements to be concise and intuitive and cleverly designed methods that benefit from well-placed `return` statements.

Limited nesting

A topic related to intuitive control flow is limited code nesting. Most introductions of new nesting levels greatly increase complexity, since the associated conditions for the respective code to be reached combine with the previous ones in often inscrutable ways. Besides being aware of the execution condition for the code he is currently reading, the reader is forced to either remember the sub-conditions introduced with each nesting level, as well as the current nesting level, or to jump back to the introduction of one or more nestings to figure out the relevant execution condition again.

Naturally, such code is far from being readable and expressive. Thus, overly deep nesting was avoided by rearranging code or using control statements like `return` in favor of opening a new `if` block. The deepest and most complicated nesting in DB2OSL has level 5 (with normal, non-nested method code having level 0), with one of these nestings being dedicated to a big enclosing `while` loop, one to a `try-catch` block and the remaining three to `if` blocks with no `else` parts and trivial one-expression conditions. Additionally, in this case all of the nesting blocks only contained a few lines of code, making the whole construction easily fit on one screen, so this was considered all right. At a few other places there occurs similar, less complicated, nesting up to level 5. TODO: references

Usage of well-known structures

Great benefit can be taken from constructs familiar to programmers regarding expressiveness. Surely, implementations based on such well-known constructs and patterns are much more likely to be instantly understood by programmers and therefore have a much higher ability of "speaking for themselves".

Examples in DB2OSL are the (extensively used) iterator concept, const correctness (see Paragraph "Const correctness" in Section 6.2.4 – Use of classes TODO), exceptions, predicates [Str13], run-time type information [Str13], helper functions [Str13] and well-known interfaces from the JAVA API like `Set` or `Collection`, as well as common JAVA constructs, like classes performing a single action (e.g. `OSLSpecPrinter`), and naming schemes, like `get.../set.../is...`.

6.2.3. Robustness against incorrect use

Care was taken to produce code that is geared to incorrect use, making it suitable for the expected environment of sporadic updates by unfamiliar and potentially even unpracticed programmers, who besides have their emphasis on the concepts of bootstrapping rather than details of the present code anyway. In fact, carefully avoiding the introduction of technical artifacts to mind, preventing programmers from focusing on the actual program logic, is an important principle of writing clean code [Str13].

In modern object-oriented programming languages, of course the main instruments for achieving this are the type system and exceptions. In particular, static type information should be used to reflect data abstraction and the "kind" of data, an object reflects, while dynamic type information should only be used implicitly, through dynamically dispatching method invocations [Str00]. Exceptions on the other hand should be used at any place related to errors and error handling, separating error handling noticeably from other code and enforcing the treatment of errors [Str13], preventing the programmer from using corrupted information in many cases.

An example of both mechanisms, static type information and exceptions, acting in combination, while cleanly fitting into the context of dynamic dispatching, are the following methods from Column.java:

```
public Boolean isNonNull()
public Boolean isUnique()
```

Their return type is the JAVA class `Boolean`, not the plain type `boolean`, because the information they return is not always known. In an early stage of the program, they returned `boolean` and were accompanied by two methods `public boolean knownIsNonNull()` and `public boolean knownIsUnique()`, telling the caller whether the respective information was known and thus the value returned by `isNonNull()` or `isUnique()`, respectively, was reliable.

They were then changed to return the JAVA class `Boolean` and to return null pointers in case the respective information is not known. This eliminated any possibility of using unreliable data in favor of generating exceptions instead, in this case a `NullPointerException`, which is thrown automatically by the JAVA Runtime Environment [Sch14] if the programmer forgets

the null check and tries to get a definite value from one of these methods when the correct value currently is not known.

Comparing two unknown values – thus, two null pointers – also yields the desired result, `true`, since the change, even when the programmer forgets that he deals with objects. However, when comparing two return values of one of the methods in general – as opposed to comparing one such return value against a constant –, errors could occur if the programmer mistakenly writes `col1.isUnique() == col2.isUnique()` instead of `col1.isUnique().booleanValue() == col2.isUnique().booleanValue()`. In this case, since the two `Boolean` objects are compared for identity [Sch14], the former comparison can return `false`, even when the two boolean values are in fact the same. However, since this case was considered much less common than cases in which the other solution could make incautious programmers introduce subtle errors, it was preferred. Besides, wrapper classes like `Boolean`, `Integer`, `Long` and `Float` are an integral part of the JAVA language [Sch14], so JAVA programmers were expected to manage to use them properly, so ultimately, since the new solution effectively prevents errors while abstaining from introducing new artifacts, it was considered fair and clean.

TODO: summary

6.2.4. Use of classes

Following the object-oriented programming paradigm [AO08], classes were heavily used to abstract from implementation details and to yield intuitively usable objects with a set of useful operations.

Identification of classes

To identify potential classes, entities from the problem domain were – if reasonable – directly represented as JAVA classes. The approach of choosing "the program that most directly models the aspects of the real world that we are interested in" to yield clean code, as described and recommended by Stroustrup [Str00], proved to be extremely useful and effective. As a consequence, the code declares classes like `Column`, `ColumnSet`, `ForeignKey`, `Table`, `TableSchema` and `SQLType`. As described in Section 6.2.2 – "Speaking code", class names were chosen to be concise but nevertheless expressive. JAVA packages were used to help attain this aim, which is why the previously mentioned class names are unambiguous. For details about package use, see Section 6.2.5 – Use of packages.

Care was taken not to introduce unnecessary classes, thereby complicating code structure and increasing the number of source files and program entities. Especially artificial classes, having little or no reference to real-world objects, could most often be avoided. On the other hand of course, it usually is not the cleanest solution to avoid such artificial classes entirely.

Section 5.3.3 – Class organization describes how the classes of DB2OSL are organized into class hierarchies.

Const correctness

Specifying in the code which objects may be altered and which shall remain constant, thus allowing for additional static checks preventing undesired modifications, is commonly referred to as "const correctness" TODO. TODO: powerful, preventing errors, clarity

Unfortunately, JAVA lacks a keyword like C++'s `const`, making it harder to achieve const correctness [TE05]. It only specifies the similar keyword `final`, which is much less expressive and doesn't allow for a similarly effective error prevention [TE05]. In particular, because `final` is not part of an object's type information, it is not possible to declare methods that return read-only objects [TE05] – placing a `final` before the method's return type would declare the method `final` [Sch14]. Similarly, there is no way to express that a method must not change the state of its object parameters. A method like `public f(final Object obj)` is only liable to not assigning a new value to its parameter object `obj` [Sch14] (which, if allowed, wouldn't affect the caller anyway [Sch14]). Methods changing its state, on the other hand, are allowed to be called on `obj` without restrictions.

Several possibilities were considered to address this problem:

- Not implementing const correctness, but stating the access rules in comments only
- Not implementing const correctness, but giving the methods which modify object states special names like `setName--USE_WITH_CARE`
- Implementing const correctness by delegating changes of objects to special "editor" objects to be obtained when an object shall be modified
- Implementing const correctness by deriving classes offering the modifying methods from read-only classes

Not implementing const correctness at all of course would have been the simplest possibility, producing the shortest and most readable code, but since incautious manipulation of objects would possibly have introduced subtle, hard-to-spot errors which in many cases would have occurred under additional conditions only and at other places, for example when inserting a `Column` into a `ColumnSet`, this method was not seriously considered.

Not implementing const correctness but using intentionally angular, conspicuous names also was not considered seriously, since it would have cluttered the code for the only sake of hopefully warning programmers of possible errors – and not attempting to avoid them technically.

So the introduction of new classes was considered the most effective and cleanest solution, either in the form of "editor" classes or derived classes offering the modifying methods directly. Again – as during the identification of classes –, the most direct solution was considered the best, so the latter form of introducing additional classes was chosen and classes like `ReadableColumn`, `ReadableColumnSet` et cetera were introduced which offer only the read-only functionality and usually occur in interfaces. Their counterparts including modifying methods also were derived from them and the implications of modifications were explained in their documentation, while the issue and the approach as such were also mentioned in the documentation of the `Readable...` classes. The `Readable...` classes can be converted to their fully-functional counterparts via downcasting (only), thereby giving a strong hint to programmers that the resulting objects are to be used with care.

Java interfaces

In Java programming, it is quiet common and often recommended [GHJV95] that every class has at least one `interface` it `implements`, specifying the operations the class provides. If no obvious `interface` exists for a class or the desired interface name is already given to some other entity, the interface is often given names like `ITableSchema` or `TableSchemaInterface`.

However, for a special purpose program with a relatively fixed set of classes mostly representing real-world artifacts from the problem domain, this approach was considered overly cluttering, introducing artificial code entities for no benefit. In particular, as explained in Section 5.3.3 – Fine structuring of db2osl, all program classes either are standing alone or belong to a class hierarchy derived from at least one interface. So, except from the standalone classes, an interface existed anyway, either "naturally" (as in the case of `Key`, for example) or because of the chosen way to implement const correctness. In some cases, these were interfaces declared in the program code, while in some cases, Java interfaces like `Set` were implemented (an obvious choice, of course, for `ColumnSet`). Introducing artificial interfaces for the standalone classes was considered unnecessary at least, if not messy.

6.2.5. Use of packages

As mentioned in Section 6.2.4 – Use of classes, class names were chosen to be concise but nevertheless expressive. This only was possible through the use of Java `packages`, which also helped structure the program.

For the current, relatively limited, extent of the program which currently comprises 45 (`public`) classes, a flat package structure was considered ideal, because it is simple and doesn't stash source files deep in subdirectories (in Java, the directory structure of the source tree is required to reflect the package structure [Sch14]). Because also every class belongs to a package, each source file is to be found exactly one directory below the root program source directory, which in many cases eases their handling.

For the description of the packages, their interaction and considerations on their structuring, see Section 5.3.2 – Coarse structuring of db2osl. For a detailed package description, refer to Appendix TODO.

Each package is documented in the source code also, namely in a file package-info.java residing in the respective package directory. This is a common scheme supported by the Eclipse IDE as well as the documentation generation systems Javadoc and Doxygen (all of which were used in the creation of the program, as described in Section 6.1 – Tools employed).

7. Summary and future work

Für den eiligen Leser ist die Vorgehensweise zusammen mit den wesentlichen Ergebnissen am Schluss in einer “Zusammenfassung” klar herauszustellen. Diese soll ausführlicher sein als die “Übersicht” am Anfang der Arbeit. Auch diese Zusammenfassung soll möglichst keine Formeln enthalten.

7.1. Summary

7.2. Future work

TODO: Software processing (and validating!) OSL

Appendices

A. Details on the db2osl implementation

A.1. Package contents (db2osl)

The following table lists the contents of each of DB2OSLs packages:

- bootstrapping
  - Bootstrapping
  - DirectMappingURIBuilder
  - URIBuilder
- cli
  - CLIDatabaseInteraction
- database
  - Column
  - ColumnSet
  - DatabaseException
  - DBSchema
  - ForeignKey
  - Key
  - PrimaryKey
  - ReadableColumn
  - ReadableColumnSet
  - ReadableForeignKey
  - ReadableKey
  - ReadablePrimaryKey
  - RetrieveDBSchema
  - Table
  - TableSchema
- helpers
  - Helpers
  - MapValueIterable
  - MapValueIterator
  - ReadOnlyIterable
  - ReadOnlyIterator
  - SQLType
  - UserAbortException
- log
  - ConsoleDiagnosticOutputHandler
  - GlobalLogger
- main
  - Main
- osl
  - OSLSpecification
- output
  - ObjectSpecPrinter
  - OSLSpecPrinter
  - SpecPrinter
- settings
  - Job
- specification
  - AttributeMap
  - EntityMap
  - IdentifierMap
  - InvalidSpecificationException
  - OBDAMap
  - OBDASpecification
  - RelationMap
  - SubtypeMap
  - TranslationTable
- test
  - CreateTestDBSchema
  - GetSomeDBSchema

Table A.1.: Class attachment to packages in DB2OSL

Bibliography

[AO08] Prince Oghenekaro Asgba and Edward E. Ogheneovo. "A Comparative Analysis of Structured and Object-Oriented Programming Methods". In: *Journal of Environmental Management* 12.4 (2008), pp. 41–46 (cit. on p. 52).

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997 (cit. on p. 48).

[BHBL09] Christian Bizer, Tom Heath, and Tim Berners-Lee. "Linked data – the story so far". In: *Semantic Services, Interoperability and Web Applications: Emerging Concepts* (2009), pp. 205–227 (cit. on p. 7).

[BL89] Tim Berners-Lee. *Information Management: A Proposal*. Tech. rep. March 1989, May 1990. CERN, 1989. URL: http://www.w3.org/History/1989/proposal.html (cit. on p. 1).

[BLF99] Tim Berners-Lee and Mark Fischetti. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. 1st. Harper San Francisco, 1999. ISBN: 0062515861 (cit. on p. 1).

[BMRSS96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. ISBN: 0471958697, 9780471958697 (cit. on p. 30).

[BRCGP04] Jesús Barrasa Rodríguez, Óscar Corcho, and Asunción Gómez-Pérez. "R_2O, an Extensible and Semantically Based Database-to-ontology Mapping Language". In: *SWDB'04: 2nd Workshop on Semantic Web and Databases*. Springer-Verlag, 2004, pp. 1069–1070 (cit. on p. 7).

[CGH+13] D. Calvanese et al. "The Optique Project: Towards OBDA Systems for Industry (Short Paper)". In: *OWL Experiences and Directions Workshop (OWLED)*. 2013 (cit. on pp. 2, 6, 7).

[Cro08] J. Crompton. *Keynote talk at the W3C Workshop on Sem. Web in Oil & Gas Industry*. 2008. URL: http://www.w3.org/2008/12/ogws-slides/Crompton.pdf (cit. on p. 6).

[Dij72] E. W. Dijkstra. *Structured Programming*. Ed. by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. London, UK, UK: Academic Press Ltd., 1972. ISBN: 0-12-200550-3 (cit. on p. 50).

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Reading, MA: Addison Wesley, 1995 (cit. on pp. 44, 54).

[Gup03] S. Gupta. *Logging in Java with the JDK 1.4 Logging API and Apache log4j*. Apresspod Series. Apress, 2003. ISBN: 9781590590997 (cit. on p. 34).

[Hee16] Dimitri van Heesch. *Doxygen: Source code documentation generator tool*. http://www.doxygen.org. [Accessed: 2016-05-13]. 2016 (cit. on p. 48).

[HPZC07] Bin He, Mitesh Patel, Zhen Zhang, and Kevin Chen-Chuan Chang. "Accessing the deep web." In: *Commun. ACM* 50.5 (2007), pp. 94–101. URL: http://dblp.uni-trier.de/db/journals/cacm/cacm50.html#HePZC07 (cit. on p. 1).

[KGJR+13] Evgeny Kharlamov et al. "Optique 1.0: Semantic Access to Big Data: The Case of Norwegian Petroleum Directorate's FactPages". In: *International Semantic Web Conference (Posters & Demos)*. Ed. by Eva Blomqvist and Tudor Groza. Vol. 1035. CEUR Workshop Proceedings. CEUR-WS.org, 2013, pp. 65–68 (cit. on pp. 6, 7).

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN: 0132350882, 9780132350884 (cit. on p. 50).

[MBSF04] J. A. Miller, G. T. Baramidze, A. P. Sheth, and P. A. Fishwick. "Investigating ontologies for simulation modeling". In: *Simulation Symposium, 2004. Proceedings. 37th Annual*. 2004, pp. 55–63. DOI: 10.1109/SIMSYM.2004.1299465 (cit. on p. 33).

[McI87] M. D. McIlroy. *A Research UNIX Reader: Annotated Excerpts from the Programmer's Manual, 1971-1986*. Tech. rep. CSTR 139. AT&T Bell Laboratories, 1987 (cit. on p. 30).

[NCFK+03] Natalya F Noy, Monica Crubézy, Ray W Fergerson, Holger Knublauch, Samson W Tu, Jennifer Vendetti, Mark A Musen, et al. "Protege-2000: an open-source ontology-development and knowledge-acquisition environment". In: *AMIA Annu Symp Proc*. Vol. 953. 2003, p. 953 (cit. on p. 33).

[Ora16] Oracle Corporation. *javadoc - The Java API Documentation Generator*. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html. [Accessed: 2016-05-13]. 2016 (cit. on p. 48).

[SAM11] Juan F. Sequeda, Marcelo Arenas, and Daniel P. Miranker. "A Completely Automatic Direct Mapping of Relational Databases to RDF and OWL". In: *International Semantic Web Conference (Posters & Demos)*. Citeseer, 2011 (cit. on p. 6).

[SAM12] Juan F. Sequeda, Marcelo Arenas, and Daniel P. Miranker. "On Directly Mapping Relational Databases to RDF and OWL". In: *Proceedings of the 21st International Conference on World Wide Web*. Lyon, France: ACM, 2012, pp. 649–658. ISBN: 978-1-4503-1229-5. DOI: 10.1145/2187836.2187924 (cit. on p. 6).

[Sch14] H. Schildt. *Java: The Complete Reference, Ninth Edition*. The Complete Reference. New York, NY, USA: McGraw-Hill Education, 2014. ISBN: 9780071808552 (cit. on pp. 34, 35, 44, 49, 51–54).

[SGH+15] Martin G. Skjæveland, Martin Giese, Dag Hovland, Espen H. Lian, and Arild Waaler. "Engineering ontology-based access to real-world data sources". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 33 (2015), pp. 112–140 (cit. on pp. 1–6, 10–12, 18, 20, 24, 35).

[SL13] Martin G Skjæveland and Espen H Lian. "Benefits of Publishing the Norwegian Petroleum Directorate's FactPages as Linked Open Data". In: *Norsk informatikkonferanse (NIK 2013). Tapir* (2013) (cit. on p. 7).

[SLH13] Martin G Skjæveland, Espen H Lian, and Ian Horrocks. "Publishing the Norwegian Petroleum Directorate's FactPages as Semantic Web Data". In: *The Semantic Web – ISWC 2013*. Springer, 2013, pp. 162–177 (cit. on p. 6).

[SSV02] Ljiljana Stojanovic, Nenad Stojanovic, and Raphael Volz. "Migrating data-intensive web sites into the semantic web". In: *Proceedings of the 2002 ACM symposium on Applied computing*. ACM. 2002, pp. 1100–1107 (cit. on p. 7).

[STCM11] Juan F. Sequeda, Syed Hamid Tirmizi, Oscar Corcho, and Daniel P. Miranker. "Survey of directly mapping SQL databases to the Semantic Web". In: *The Knowledge Engineering Review* 26.04 (2011), pp. 445–486 (cit. on p. 6).

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. 3rd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201700735 (cit. on pp. 2, 39, 51, 52).

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Boston, MA, USA: Addison-Wesley Professional, 2013. ISBN: 0321563840, 9780321563842 (cit. on pp. 44, 49, 51).

[TE05] Matthew S. Tschantz and Michael D. Ernst. "Javari: Adding Reference Immutability to Java". In: *SIGPLAN Not.* 40.10 (Oct. 2005), pp. 211–230. ISSN: 0362-1340. DOI: 10.1145/1103845.1094828 (cit. on p. 53).

[W3C09] W3C XML Core Working Group. *XML Base (Second Edition)*. https://www.w3.org/TR/xmlbase/. [Accessed: 2016-04-02]. 2009 (cit. on p. 24).

[W3C12] W3C OWL Working Group. *OWL 2 Web Ontology Language - Document Overview (Second Edition)*. https://www.w3.org/TR/owl2-overview/. [Accessed: 2016-04-02]. 2012 (cit. on p. 24).

[W3C14] W3C RDF Working Group. *RDF 1.1 Concepts and Abstract Syntax*. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/. [Accessed: 2016-05-13]. 2014 (cit. on pp. 1, 5).

[W3CR12a] W3C RDB2RDF Working Group. *A Direct Mapping of Relational Data to RDF*. https://www.w3.org/TR/rdb-direct-mapping/. [Accessed: 2016-04-06]. 2012 (cit. on pp. 1, 8, 9, 18).

[W3CR12b] W3C RDB2RDF Working Group. *R2RML: RDB to RDF Mapping Language*. https://www.w3.org/TR/r2rml/. [Accessed: 2016-05-20]. 2012 (cit. on pp. 9, 12).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature