Integrating Relational Databases with the Semantic Web: A Reflection

Juan F. Sequeda(✉)

Capsenta, Austin, USA
juan@capsenta.com

Abstract. From the beginning it was understood that the success of the Semantic Web hinges on integrating the vast amount of data stored in Relational Databases. This manuscript reflects on the last 10 years of our research results to integrate Relational Databases with the Semantic Web. Since 2007, our research has led us to answer the following question: *How and to what extent can Relational Databases be Integrated with the Semantic Web?* The answer comes in two parts. We start by presenting how to get from Relational Databases to the Semantic Web via mappings, such as the W3C Direct Mapping and R2RML standards. Subsequently, we present how the Semantic Web can access Relational Databases. We finalize with how Relational Databases and Semantic Web technologies are being used practice for data integration and discuss open challenges.

1 Introduction

The success of the Semantic Web hinges on integrating the vast amount of data stored in Relational Databases. We have gone a long way in the past 10 years. As of 2017, a successful repeated use case for Relational Databases and the Semantic Web is to address data integration needs. Such systems are now being deployed in industrial applications. So, how did we get here? The goal of this manuscript is to reflect on the last 10 years of our research results to integrate Relational Databases with the Semantic Web [64].

In 2007, we began investigating the relationship between Relational Databases and the Semantic Web. Specifically, the research question was the following: *How and to what extent can Relational Databases be integrated with the Semantic Web?* The thesis is that much of the existing Relational Database infrastructure can be reused to support the Semantic Web.

In the first part, we describe how to get from Relational Databases to the Semantic Web via mappings. Starting with a 2007 workshop, titled "RDF Access to Relational Databases"[1], the W3C sponsored a series of activities to address this issue. At that workshop, the acronym, RDB2RDF, Relational Database to Resource Description Framework, was coined. In September 2012, these activities culminated in the ratification of two W3C standards, colloquially known as Direct Mapping [7] and R2RML [25].

[1] http://www.w3.org/2007/03/RdfRDB/.

G. Ianni et al. (Eds.): Reasoning Web 2017, LNCS 10370, pp. 68–120, 2017.
DOI: 10.1007/978-3-319-61033-7_4

By design, both these standards avoid any content that speaks about implementation, directly or indirectly. The standards concern is syntactic transformation of the contents of rows in relational tables to RDF. The R2RML language includes statements that specify which columns and tables are mapped to properties and classes of a domain ontology. Thus, the language empowers a developer to examine the contents of a relational database and write a mapping specification. Furthermore, we present an extended Direct Mapping which address some shortcomings of the W3C Direct Mapping and study it with respect to two fundamental (information and query preservation) and two desired (monotonicity and semantics preservation) properties.

In the second part, we describe the opposite direction, how the Semantic Web can access Relational Databases. Once a mapping has been defined, let it be a Direct Mapping or a user defined R2RML mappings, the goal is to evaluate SPARQL queries against the Relational Database. These contributions are embodied in our system called Ultrawrap. We identified two existing relational query optimizations in commercial Relational Databases, detection of unsatisfiable conditions and self-join elimination which are used for SPARQL execution. Empirical analysis consistently yield that SPARQL query execution performance on Ultrawrap is comparable to that of SQL queries written directly for the relational representation of the data. Furthermore, we present a method for Relational Databases to support inheritance and transitivity by compiling the ontology as mappings, implementing the mappings as SQL views, using SQL recursion and optimizing by materializing a subset of views. This approach was implemented as an extension of Ultrawrap to support the Ontology-Based Data Access paradigm. Empirical analysis reveals that Relational Databases are able to effectively act as reasoners.

To understand the relationship between Relational Databases and the Semantic Web, we adopt a methodology where we first start small. That is why we first studied a simple mapping which is the Direct Mapping. Subsequently we studied how to accomplish SPARQL to SQL rewriting under the direct mapping. After the direct mapping relationship was understood, we continued our work with customized mappings represented in R2RML and reasoning.

We highlight two on-going challenges when Relational Databases and Semantic Web technologies are combined for data integration in the real world: ontology and mapping engineering. We argue for the need of a pay-as-you-go methodology to create mappings and ontologies. We close with a set of open problems.

2 Preliminaries

This sections presents the notation and definitions used throughout this manuscript. We define the three standards comprising Semantic Web: RDF, the graph data model; OWL, the ontology language; and SPARQL, the query language for RDF. Subsequently, the expressivity of the OWL dialect used in this research is presented. For more detailed preliminaries, we refer the reader to Chap. 2 of [64]

2.1 Running Example

Throughout this manuscript, we use the data illustrated in Fig. 1 as a running example. The precise corresponding SQL statements are:

```
CREATE TABLE order (
  orderid INT PRIMARY KEY,
  date DATE,
  total FLOAT,
  currency VARCHAR(50),
  status INT
)

CREATE TABLE lineitem (
  lineid INT PRIMARY KEY,
  price FLOAT,
  quantity INT,
  product VARCHAR(50),
  orderid INT,
  FOREIGN KEY(orderid) REFERENCES ORDER(orderid)
)
```

orderid	date	total	currency	status
1234	2017-04-15	100	USD	1

lineid	price	quantity	product	orderid
6789	30	2	Foo	1234
6790	20	2	Bar	1234

Fig. 1. SQL used to create the running example

2.2 Relational Databases

A database is a collection of data. A Relational Database is a database founded on the relational model. The relational model represents data in terms of tuples (rows), grouped into relations (tables). Relational Algebra is used as a query language for Relational Databases.

Because nulls appear in practice in RDBMS, it is important to present a formal definition of Relational Databases with respect to null values. Assume, a countably infinite domain $\mathbf{D}$ of constants and a reserved symbol `NULL` that is not in $\mathbf{D}$. A *database schema* $\mathbf{R}$ is a finite set of relation names, where for each $R \in \mathbf{R}$, $att(R)$ denotes the nonempty finite set of attribute names associated with R. The arity of R, denoted as $arity(R)$, is the number of elements of the set $att(R)$. An instance I of $\mathbf{R}$ assigns to each relation symbol $R \in \mathbf{R}$, a finite set of tuples $R^I = \{t_1, \ldots, t_\ell\}$. Each tuple t_j $(1 \leq j \leq \ell)$ is a function that

assigns to each attribute in $att(R)$ a value from $(\mathbf{D} \cup \{\texttt{NULL}\})$, denoted as $t : att(R) \rightarrow (\mathbf{D} \cup \{\texttt{NULL}\})$. The value of an attribute A in a tuple t_j is denoted by $t_j.A$. Moreover, $R(t_j)$ is a fact in I if $t_j \in R^I$. The notation $R(t_j) \in I$ is used in this case. We also view instances as sets of facts.

Relational Algebra consists of operators which take one or two relations as operands and produce one relation as a result. The basic operators of relational algebra are: selection, projection, rename, join, union and difference. Selection selects tuples from a relation satisfying a condition. Projection chooses subset of the attributes of a relation. Rename allows to change the name of an attribute. Join combines two relations into one on the basis of a condition. Union is the relation containing all tuples from both relations. Difference is the relation containing all tuples of the first relation that do not appear in the second relation. Relational Algebra operators can be composed into relational algebraic expressions. These relational algebraic expressions are then used to formulate queries over a Relational Database.

Recall that Relational Databases containing null values are considered. For full details on the syntax and semantics of Relational Algebra where null values play a role, we refer the reader to Chap. 2 of [64].

2.3 Semantic Web

The Semantic Web is an extension to the Web that enables intelligent access to data on the Web. The technologies supporting the Semantic Web consist of a set of standards: RDF as the graph data model, OWL as the ontology language, and SPARQL as the query language.

RDF: RDF stands for Resource Description Framework, which is a framework for representing information about resources in the Web. By resource, we mean anything in the world including physical things, documents, abstract concepts, etc[2]. RDF considers three types of values: resource identifiers (IRIs) to denote resources, literals to denote values such as strings, and blank nodes to denote the existence of unnamed resources which are existentially quantified variables that can be used to make statements about unknown (but existent) resources.

Assume there are pairwise disjoint infinite sets $\mathbf{I}$ (IRIs), $\mathbf{B}$ (blank nodes) and $\mathbf{L}$ (literals). A tuple $(s, p, o) \in (\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$ is called an RDF triple, where s is the subject, p is the predicate and o is the object. A finite set of RDF triples is called an RDF graph. Assume that `triple` is a ternary predicate that stores RDF graphs in the obvious way: every triple $(a, b, c) \in G$ is stored as $\texttt{triple}(a, b, c)$. Moreover, assume the existence of an infinite set $\mathbf{V}$ of variables disjoint from the above sets, and assume that every element in $\mathbf{V}$ starts with the symbol "?".

[2] The term "entity" can be considered synonymous to resource.

Example 1. Consider representing the statement "There is a person whose name is Juan Sequeda" in RDF. This can be represented with two RDF triples. The first RDF triple

$$\texttt{triple}(\texttt{http://juansequeda.com\#me}, \texttt{type}, \texttt{foaf:Person})$$

states that the resource identified by http://juansequeda.com#me *is of type Person. The type relationship is represented with* `rdf:type`*. Additionally, the concept Person is identified by the IRI* `foaf:Person`*. Note that* `rdf:` *and* `foaf:` *are being used instead of a full IRI. These are prefixes that replace a part of the IRI*[3]*. The second RDF triple*

$$\texttt{triple}(\texttt{http://juansequeda.com\#me}, \texttt{foaf:name}, \texttt{"Juan Sequeda"})$$

states that http://juansequeda.com#me *has a name which is "Juan Sequeda". The concept of name is identified by the IRI* `foaf:name`*.*

OWL: OWL stands for Web Ontology Language, which is the language to represent ontologies on the Web. In order to define the notion of ontology, the following set of reserved keywords are defined as **O**: {`subClass`, `subProp`, `dom`, `range`, `type`, `equivClass`, `equivProp`, `inverse`, `symProp`, `transProp`}.

Assume that $\mathbf{O} \subseteq \mathbf{I}$. Two types of RDF triples are distinguished: ontological and assertional. Ontological RDF triples define the ontology. Assertional RDF triples define the facts. The formal definitions are the following:

Definition 1 (Ontological RDF Triple). *Following the definition presented by Weaver and Hendler [75], an RDF triple* (a, b, c) *is ontological if:*

1. $a \in (\mathbf{I} \setminus \mathbf{O})$*, and*
2. *either* $b \in (\mathbf{O} \setminus \{\texttt{type}\})$ *and* $c \in (\mathbf{I} \setminus \mathbf{O})$*, or* $b =$ `type` *and* c *is either* `symProp` *or* `transProp`*.*

In other words, an ontological RDF triple will always have as a subject an element in **I** but not in **O**. There are two types of ontological RDF triples. First, the predicate is an element in **O** but not `type` and the object is an element in **I** but not in **O**. Second, if the predicate is `type`, then the object is either `symProp` or `transProp`.

Definition 2 (Assertional RDF Triple). *An RDF triple* (a, b, c) *is assertional if it is not ontological.*

Definition 3 (Ontology). *An ontology* $\mathcal{O}$ *is defined as a finite set of ontological RDF triples.*

[3] The prefix "`rdf:`" represents http://www.w3.org/1999/02/22-rdf-syntax-ns#, hence the full IRI for `rdf:type` is http://www.w3.org/1999/02/22-rdf-syntax-ns#type. Additionally, the prefix "`foaf:`" represents http://xmlns.com/foaf/0.1/, hence the full IRI for `foaf:Person` is http://xmlns.com/foaf/0.1/Person.

The semantics of an ontology $\mathcal{O}$ is usually defined by representing it as a set of description logic axioms, and then relying on the semantics of the logic [10] (which, in turn, is derived from the semantics of first-order logic). It is more convenient to directly define a set of first-order formulae, denoted as $\Sigma_{\mathcal{O}}$, to encode the ontology $\mathcal{O}$. The semantics of each ontological triple of an ontology, $t \in \mathcal{O}$, is defined as a first-order formula φ_t over the predicate `triple`. Definitions 4–12 presents the first-order formula for ontological triples. Finally, the set $\Sigma_{\mathcal{O}}$ of first-order formulae encoding the ontology $\mathcal{O}$ is define as $\{\varphi_t \mid t \in \mathcal{O}\}$.

Definition 4 (Subclass). If a is a subclass of b and x is an instance of a, then x is an instance of b. The first-order formula is:

$$\varphi_{(a,\texttt{subClass},b)} = \forall x \left(\texttt{triple}(x, \texttt{type}, a) \rightarrow \texttt{triple}(x, \texttt{type}, b)\right)$$

Definition 5 (Subproperty). If a is a subproperty of b, then all pairs of resources (x, y) which are related by a are also related by b. The first-order formula is:

$$\varphi_{(a,\texttt{subProp},b)} = \forall x \forall y \left(\texttt{triple}(x, a, y) \rightarrow \texttt{triple}(x, b, y)\right)$$

Definition 6 (Domain). If a has a domain b then any resource x that is related to a is an instance of b. The first-order formula is:

$$\varphi_{(a,\texttt{dom},b)} = \forall x \forall y \left(\texttt{triple}(x, a, y) \rightarrow \texttt{triple}(x, \texttt{type}, b)\right)$$

Definition 7 (Range). If a has a range b then any resource y that is related to a is an instance of b. The first-order formula is:

$$\varphi_{(a,\texttt{range},b)} = \forall x \forall y \left(\texttt{triple}(x, a, y) \rightarrow \texttt{triple}(y, \texttt{type}, b)\right)$$

Definition 8 (Equivalent Class). If a has an equivalent class of b and x is an instance of a, then x is an instance of b. Conversely, if x is an instance of b, then x is an instance of a. The first-order formula is:

$$\varphi_{(a,\texttt{equivClass},b)} = \forall x \left(\texttt{triple}(x, \texttt{type}, a) \leftrightarrow \texttt{triple}(x, \texttt{type}, b)\right)$$

Definition 9 (Equivalent Property). If a has an equivalent property of b, then all pairs of resources (x, y) which are related by a are also related by b. Conversely, all pairs of resources (x, y) which are related by b are also related by a. The first-order formula is:

$$\varphi_{(a,\texttt{equivProp},b)} = \forall x \forall y \left(\texttt{triple}(x, a, y) \leftrightarrow \texttt{triple}(x, b, y)\right)$$

Definition 10 (Inverse Property). If a has an inverse property of b, then all pairs of resources (x, y) which are related by a are also related by b by the pair (y, x). Conversely, all pairs of resources (y, x) which are related by b are also related by a by the pair (x, y) The first-order formula is:

$$\varphi_{(a,\texttt{inverse},b)} = \forall x \forall y \left(\texttt{triple}(x, a, y) \leftrightarrow \texttt{triple}(y, b, x)\right)$$

Definition 11 (Symmetric Property). If a is a symmetric property, then all pairs of resources (x, y) which are related by a are also related as the pair (y, x). The first-order formula is:

$$\varphi_{(a,\texttt{type},\texttt{symProp})} = \forall x \forall y \, (\texttt{triple}(x, a, y) \rightarrow \texttt{triple}(y, a, x))$$

Definition 12 (Transitive Property). If a is a transitive property, and for all pairs of resources (x, y) and (y, z) which are related by a then the pair (x, z) is also related by a. The first-order formula is:

$$\varphi_{(a,\texttt{type},\texttt{transProp})} = \forall x \forall y \forall z \, (\texttt{triple}(x, a, y) \wedge \texttt{triple}(y, a, z) \rightarrow \texttt{triple}(x, a, z))$$

Given that the semantics of an ontology $\mathcal{O}$ has been defined as set of first order logic formulae $\Sigma_{\mathcal{O}}$ and a RDF graph G using the predicate `triple`, then $\Sigma_{\mathcal{O}} \cup G$ is consistent (and inconsistent) in the usual sense of First Order Logic.

Example 2 The following ontology states that an Executive and ITEmployee are both Employees. Additionally that the property hasSuperior is a transitive relationship from an Employee to another Employee.

`triple(`:Executive, subClass, :Employee`)`
`triple(`:Programmer, subClass, :ITEmployee`)`
`triple(`:SysAdmin, subClass, :ITEmployee`)`
`triple(`:ITEmployee, subClass, :Employee`)`
`triple(`:hasSuperior, type, transProp`)`
`triple(`:hasSuperior, dom, :Employee`)`
`triple(`:hasSuperior, range, :Employee`)`

Ontology Profiles. The expressiveness of an ontology language can be specified by profiles. The Semantic Web technology stack specifies four ontology profiles: RDFS, OWL 2 EL, OWL 2 QL and OWL 2 RL [13,50].

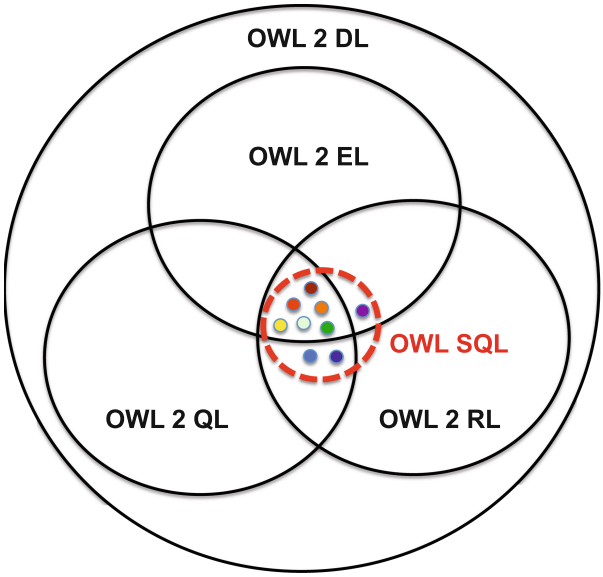
RDF Schema (RDFS) extends RDF as a schema language for RDF and a lightweight ontology language [13]. It includes constructs to declare classes, hierarchies between classes and properties and relate the domain and range of a property to a certain class. Ontological triples with `subClass`, `subProp`, `dom`, `range`, `type`, `equivClass`, `equivProp` are in this profile. The following three profiles, OWL 2 EL, QL and RL, extend the expressiveness of RDFS.

OWL 2 EL profile is used to represent ontologies that define very large numbers of classes and/or properties with transitivity. This language has been tailored to model large life science ontologies, while still supporting efficient reasoning. OWL 2 EL is based on the EL++ Description Logic [9]. Ontological triples with `transProp` are in this profile.

OWL 2 QL provides constructs to express conceptual models such as UML class diagrams and ER diagrams. This language was designed so that data that is stored in a standard relational database system can be queried through an ontology via rewriting mechanisms. OWL 2 QL is based on the DL-Lite family of description logics [16]. Ontological triples with `inverse` and `symProp` are in this profile.

OWL 2 RL provides constructs to represent rules in ontologies. This language has been tailored for rule-based reasoning engines. OWL 2 RL is based on Description Logic Programs (DLP) [35]. Ontological triples with `inverse` and `symProp` are also in this profile.

The ontology expressivity considered in this work (as defined in Definitions 4–12) is not specific to a single OWL profile. Thus, we propose a new ontology profile, OWL-SQL, which expresses the types of ontologies considered in this dissertation. Figure 2 denotes the expressivity of OWL-SQL with respect to the OWL 2 EL, QL and RL profiles.



	EL	QL	RL
subClass	X	X	X
subProp	X	X	X
domain	X	X	X
range	X	X	X
eqClass	X	X	X
eqProp	X	X	X
inverseProp		X	X
symProp		X	X
transProp	X		X

Fig. 2. OWL-SQL, proposed OWL profile

The expressivity of OWL-SQL is subsumed by early ontology profile proposals known as RDFS-Plus [4], OWL-LD [32] and RDFS 3.0 [39].

2.4 SPARQL

SPARQL is the standard query language for RDF [38,59]. SPARQL is a graph pattern matching query language and has a syntax similar to SQL. A SPARQL query contains a set of triple patterns called basic graph patterns. Triple patterns are similar to RDF triples with the exception that the subject, predicate or object can be variables (denoted by a leading question mark "?"). The answer of a SPARQL query P over an RDF graph G is a finite set of *mappings*, where a mapping μ is a partial function from the set $\mathbf{V}$ of variables to $(\mathbf{I} \cup \mathbf{L} \cup \mathbf{B})$[4].

[4] Recall that $\mathbf{V}$ is an infinite set of variables disjoint from $\mathbf{I}$, $\mathbf{B}$ and $\mathbf{L}$ and that every element in $\mathbf{V}$ starts with the symbol "?". See Sect. 2.3.

Example 3. Consider the RDF triples in Example 1. The following SPARQL query asks for all names of people.

```
SELECT ?n
WHERE {
  ?s rdf:type foaf:Person.
  ?s foaf:name ?n.
}
```

The basic graph pattern consists of two triple patterns. Matching these triple patterns with the RDF triples gives the answer `"Juan Sequeda"`.

The semantics of SPARQL is defined as a function $[\![\cdot]\!]_G$ that, given an RDF graph G, takes a graph pattern expression and returns a set of mappings. The reader is referred to [64] for more detail.

3 From Relational Databases to the Semantic Web: Mappings

3.1 W3C Direct Mapping

The W3C Direct Mapping [7] is an automatic approach of translating a relational database to RDF. The W3C Direct Mapping takes as input a relational database (data and schema), and generates an RDF graph that is called the direct graph. No additional user input is needed to map the relational data to RDF. The structure of the resulting RDF graph directly reflects the structure of the database. The RDF vocabulary is automatically generated from the names of database schema elements. Neither the structure nor the vocabulary can be changed. If needed, the resulting RDF graph can be transformed further by the user using other RDF to RDF mapping approaches such as SPARQL CONSTRUCT.

The W3C Direct Mapping consists of two parts. A specification to generate identifiers for a table, column foreign key and rows and a specification using the identifiers, in order to generate the direct graph.

Generating Identifiers. The W3C Direct Mapping generates an identifier for rows, tables, columns and foreign keys. If a table has a primary key, then the row identifier will be an IRI, otherwise a blank node. The identifiers for tables, columns and foreign keys are IRIs. It is important to note that in this paper we present relative IRIs which must be resolved by appending to a given base IRI. Throughout this document, http://ex.com/rdb2rdf/ is the base IRI. All strings are percent encoded in order to generate a safe IRI[5].

If a table has a primary key, then the row identifier will be an IRI, obtained by concatenating the base IRI, the percent-encoded form of the table name, the '#' character and for each column in the primary key, in order:

[5] For example, a space is replaced with %20 e.g., the percent encoding of "Hello World" is "Hello%20World".

– the percent-encoded form of the column name,
– the '=' character
– the percent-encoded lexical form of the canonical RDF literal representation of the column value
– if it is not the last column in the primary key, the ';' character

For example the IRI for the row of the order table is <http://ex.com/rdb2rdf/order#orderid=1234>. If a table does not have a primary key, then the row identifier is a fresh blank node that is unique to each row

The IRI for a table is obtained by concatenating the base IRI with the percent-encoded form of the table name. For example the table IRI of the order table is <http://ex.com/rdb2rdf/order> The IRI for an attribute is obtained by concatenating the base IRI with the percent-encoded form of the table name, the '#' character and the percent-encoded form of the column name. For example, the Literal Property IRI of the date attribute of the order table is <http://ex.com/rdb2rdf/order#date> Finally the IRI for foreign key is obtained by concatenating the base IRI with the percent-encoded form of the table name, the string '#ref-' and for each column in the foreign key, in order:

– the percent-encoded form of the column name,
– if it is not the last column in the foreign key, a ';' character

For example, the reference Property IRI of the foreign key orderid of the lineitem table is <http://ex.com/rdb2rdf/lineitem#ref-orderid>

Generating the Direct Graph. A Direct Graph is the RDF graph resulting from directly mapping each of the rows of each table and view in a database schema. Each row in a table generates a Row Graph. The row graph is an RDF graph consisting of the following triples: (1) a row type triple, (2) a literal triple for each column in a table where the column value is non-NULL and (3) a reference triple for each foreign key in the table where none of the column values is NULL. A row type triple is an RDF triple with the subject as the row node for the row, the predicate as the RDF IRI `rdf:type` and the object as the table IRI for the table name. A literal triple is an RDF triple with the subject as the row node for the row, the predicate as the literal property IRI for the column and the object as the natural RDF literal representation of the column value. Finally, a reference triple is an RDF triple with the subject as the row node for the row, the predicate as the reference property IRI for the columns and the object as the row node for the referenced row.

Example 4 (W3C Direct Mapping of Running Example). RDF generated by the W3C Direct Mapping of the running example, in Turtle syntax. Recall that the IRIs in the example are relative IRIs which must be resolved by appending to the base IRI http://ex.com/rdb2rdf/.

```
<order#orderid=1234> rdf:type <order> ;
    <order#orderid>"1234" ;
```

```
    <order#date> "2017-04-15";
    <order#total> "100";
    <order#currency> "USD";
    <order#status> "1".
<lineitem#lineid=6789> rdf:type <lineitem>;
    <lineitem#lineid> "6789";
    <lineitem#price> "30";
    <lineitem#quantity> "2";
    <lineitem#product> "Foo";
    <lineitem#orderid> "1234";
    <lineitem#ref-orderid> <order#orderid=1234>.
<lineitem#lineid=6790> rdf:type <lineitem>;
    <lineitem#lineid> "6790";
    <lineitem#price> "20";
    <lineitem#quantity> "2";
    <lineitem#product> "Bar";
    <lineitem#orderid> "1234";
    <lineitem#ref-orderid> <order#orderid=1234>.
```

The formal semantics of the W3C Direct Mapping has been defined in Datalog. We refer the reader to the W3C Direct Mapping standard document for details [7]. The left hand side of each rule is the RDF Triple output. The right hand side of each rule consists of a sequence of predicates from the relational database and built-in predicates.

3.2 $\mathcal{DM}$: Direct Mapping as Ontology

The W3C Direct Mapping standard has two main shortcomings. First, the mapping is only from relational data to RDF data. The relational schema is not taken in account. Second, the semantics of the W3C Direct Mapping is not defined for NULL values as described in the specification: *"The direct mapping does not generate triples for NULL values. Note that it is not known how to relate the behavior of the obtained RDF graph with the standard SQL semantics of the NULL values of the source RDB."* In this section, we first formally introduce the notion of a direct mapping. Subsequently we introduce a new Direct Mapping which addresses the aforementioned shortcomings.

A direct mapping is a default way to translate relational databases into RDF (without any input from the user on how the relational data should be translated). The input of a direct mapping $\mathcal{M}$ is a relational schema $\mathbf{R}$, a set Σ of PKs (Primary Keys) and FKs (Foreign Keys) over $\mathbf{R}$ and an instance I of $\mathbf{R}$. The output is an RDF graph with OWL vocabulary.

Assume $\mathcal{G}$ is the set of all RDF graphs and $\mathcal{RC}$ is the set of all triples of the form $(\mathbf{R}, \Sigma, I)$ such that $\mathbf{R}$ is a relational schema, Σ is a set of PKs and FKs over $\mathbf{R}$ and I is an instance of $\mathbf{R}$.

Definition 13 (Direct Mapping). *A direct mapping $\mathcal{M}$ is a total function from $\mathcal{RC}$ to $\mathcal{G}$.*

We introduce the Direct Mapping as Ontology [65], denoted as $\mathcal{DM}$, which extends the W3C Direct Mapping [7] and combines with a direct mapping of relational database schema to an OWL ontology [69,73]. Additionally, $\mathcal{DM}$ considers the case when the input database has NULL values. $\mathcal{DM}$ is defined as a set of Datalog predicate and rules[6].

1. Five predicates that encode the input relational schema and instance to $\mathcal{DM}$: $\textsc{Rel}(r)$: Indicates that r is a relation name in $\mathbf{R}$, $\textsc{Attr}(a, r)$: Indicates that a is an attribute in the relation r in $\mathbf{R}$, $\text{PK}_n(a_1, \ldots, a_n, r)$: Indicates that $r[a_1, \ldots, a_n]$ is a primary key in Σ, $\text{FK}_n(a_1, \ldots, a_n, r, b_1, \ldots, b_n, s)$: Indicates that $r[a_1, \ldots, a_n] \subseteq_{\text{FK}} s[b_1, \ldots, b_n]$ is a foreign key in Σ, and $\textsc{Value}(v, a, t, r)$ which Indicates that v is the value of an attribute a in a tuple with identifier t in a relation r (that belongs to $\mathbf{R}$).
2. Three predicates that are used to store an ontology: $\textsc{Class}(c)$ indicates that c is a class; $\text{OP}_n(p_1, \ldots, p_n, d, r)$ indicates that $p_1, \ldots, p_n$ $(n \geq 1)$ form an object property with domain d and range r and $\text{DTP}(p, d)$ indicates that p is a data type property with domain d.
3. Twelve Datalog rules that generate a putative ontology from a relational schema. The rules can be summarized as follows: a table is translated to an OWL Class unless the table represents a binary relationship, then it is translated to an OWL Object Property. Foreign Keys are translated to OWL Object Properties while attributes are translated to OWL Datatype Properties.
4. Ten Datalog rules that generate the OWL ontology from the predicates that are used to store an ontology which include rules to generate IRIs and express the ontology as RDF triples.
5. Ten Datalog rules that generate RDF triples from a relational instance based on the putative ontology.

We present example Datalog rules for the generation of classes and datatype properties. We refer the reader to [65] for the detailed list of Datalog rules. A class, defined by the predicate $\textsc{Class}$, is any relation that is not a binary relation. A relation R is a binary relation, defined by the predicate $\textsc{BinRel}$, between two relations S and T if (1) both S and T are different from R, (2) R has exactly two attributes A and B, which form a primary key of R, (3) A is the attribute of a foreign key in R that points to S, (4) B is the attribute of a foreign key in R that points to T, (5) A is not the attribute of two distinct foreign keys in R, (6) B is not the attribute of two distinct foreign keys in R, (7) A and B are not the attributes of a composite foreign key in R, and (8) relation R does not have incoming foreign keys. The formal definition of $\textsc{BinRel}$ can be found in [65]. Therefore, the predicate $\textsc{Class}$ is defined by the following Datalog rules:

$$\textsc{Class}(X) \leftarrow \textsc{Rel}(X), \neg\textsc{IsBinRel}(X)$$
$$\textsc{IsBinRel}(X) \leftarrow \textsc{BinRel}(X, A, B, S, C, T, D)$$

For instance, we have that $\textsc{Class}(\texttt{order})$ holds in our example.

[6] We refer the reader to [2] for the syntax and semantics of Datalog.

Every attribute in a non-binary relation is mapped to a data type property, defined by the predicate DTP, which is defined by the following Datalog rule:

$$\text{DTP}(A, R) \leftarrow \text{Attr}(A, R), \neg\text{IsBinRel}(R)$$

For instance, we have that DTP(`date`, `order`) holds in our example.

We now briefly define the rules that translates a relational database schema into an OWL vocabulary. We introduce a family of rules that produce IRIs for classes and data type properties identified by the mapping (which are stored in the predicates Class and DTP). Note that the IRIs generated can be later on replaced or mapped to existing IRIs available in the Semantic Web. Assume given a base IRI `base` for the relational database to be translated (for example, "http://ex.com/rdb2rdf/"), and assume a family of built-in predicates Concat_n ($n \geq 2$) is given, such that Concat_n has $n + 1$ arguments and $\text{Concat}_n(x_1, \ldots, x_n, y)$ holds if y is the concatenation of the strings x_1, ..., x_n. Then by following the approach proposed in [7], $\mathcal{DM}$ uses the following Datalog rules to produce IRIs for classes and data type properties:

$$\begin{aligned} \text{ClassIRI}(R, X) &\leftarrow \text{Class}(R), \text{Concat}_2(\texttt{base}, R, X) \\ \text{DTP\_IRI}(A, R, X) &\leftarrow \text{DTP}(A, R), \text{Concat}_4(\texttt{base}, R, \texttt{"\#"}, A, X) \end{aligned}$$

For instance, http://ex.com/rdb2rdf/order is the IRI for the `order` relation in our example, and http://ex.com/rdb2rdf/order#date is the IRI for attribute `date` in the `order` relation.

The following Datalog rules are used to generate the RDF representation of the OWL vocabulary. A rule is used to collect all the classes:

$$\begin{aligned} \text{Triple}(U, \texttt{"rdf:type"}, \texttt{"owl:Class"}) \leftarrow \\ \text{Class}(R), \text{ClassIRI}(R, U) \end{aligned}$$

The predicate Triple is used to collect all the triples of the RDF graph generated by the direct mapping $\mathcal{DM}$. The following rule is used to collect all the data type properties:

$$\begin{aligned} \text{Triple}(U, \texttt{"rdf:type"}, \texttt{"owl:DatatypeProperty"}) \leftarrow \\ \text{DTP}(A, R), \text{DTP\_IRI}(A, R, U) \end{aligned}$$

The following rule is used to collect the domains of the data type properties:

$$\begin{aligned} \text{Triple}(U, \texttt{"rdfs:domain"}, W) \leftarrow \\ \text{DTP}(A, R), \text{DTP\_IRI}(A, R, U), \text{ClassIRI}(R, W) \end{aligned}$$

Example 5 (Direct Mapping as Ontology of Running Example). OWL generated by the Direct Mapping as Ontology of the running example, in Turtle syntax. The RDF triples from the Direct Mapping as Ontology are the same as in Example 4. Recall that the IRIs in the example are relative IRIs which must be resolved by appending to the base IRI http://ex.com/rdb2rdf/.

```
<order> rdf:type owl:Class.
<order#orderid> rdf:type owl:DatatypeProperty ;
    rdfs:domain <order>.
<order#date> rdf:type owl:DatatypeProperty;
    rdfs:domain <order>.
<order#total> rdf:type owl:DatatypeProperty;
    rdfs:domain <order>.
<order#currency> rdf:type owl:DatatypeProperty;
    rdfs:domain <order>.
<order#status> rdf:type owl:DatatypeProperty;
    rdfs:domain <order>.
<lineitem> rdf:type owl:Class.
<lineitem#lineid> rdf:type owl:DatatypeProperty;
    rdfs:domain <lineitem>.
<lineitem#price> rdf:type owl:DatatypeProperty;
    rdfs:domain <lineitem>.
<lineitem#quantity> rdf:type owl:DatatypeProperty;
    rdfs:domain <lineitem>.
<lineitem#product> rdf:type owl:DatatypeProperty;
    rdfs:domain <lineitem>.
<lineitem#orderid> rdf:type owl:DatatypeProperty;
    rdfs:domain <lineitem>.
<lineitem#ref-pid> rdf:type owl:ObjectProperty;
    rdfs:domain <lineitem>;
    rdfs:range <order>.
```

Direct Mapping Properties. We study two properties that are fundamental to a direct mapping: information preservation and query preservation. Additionally we study two desirable properties: monotonicity and semantics preservation.

A direct mapping is information preserving if it does not lose any information about the relational instance being translated, that is, if there exists a way to recover the original database instance from the RDF graph resulting from the translation process. Formally, assuming that $\mathcal{I}$ is the set of all possible relational instances, we have that:

Definition 14 (Information Preservation). *A direct mapping $\mathcal{M}$ is* information preserving *if there is a computable mapping $\mathcal{N} : \mathcal{G} \to \mathcal{I}$ such that for every relational schema $\mathbf{R}$, set Σ of PKs and FKs over $\mathbf{R}$, and instance I of $\mathbf{R}$ satisfying Σ: $\mathcal{N}(\mathcal{M}(\mathbf{R}, \Sigma, I)) = I$.*

Recall that a mapping $\mathcal{N} : \mathcal{G} \to \mathcal{I}$ is computable if there exists an algorithm that, given $G \in \mathcal{G}$, computes $\mathcal{N}(G)$.

Theorem 1. *The direct mapping $\mathcal{DM}$ is information preserving.*

The proof of this theorem is straightforward, and it involves providing a computable mapping $\mathcal{N} : \mathcal{G} \to \mathcal{I}$ that satisfies the condition in Definition 14, that

is, a computable mapping $\mathcal{N}$ that can reconstruct the initial relational instance from the generated RDF graph.

A direct mapping is query preserving if every query over a relational database can be translated into an equivalent query over the RDF graph resulting from the mapping. That is, query preservation ensures that every relational query can be evaluated using the mapped RDF data.

I define query preservation, we focus on relational queries Q that can be expressed in relational algebra [2] and RDF queries $Q^\star$ that can be expressed in SPARQL [55,59]. Given the mismatch in the formats of these query languages (null can appear as a result of a relational query while null does not in a SPARQL query), we introduce a function *tr* that converts tuples returned by relational algebra queries into mappings returned by SPARQL. Formally, given a relational schema $\mathbf{R}$, a relation name $R \in \mathbf{R}$, an instance I of $\mathbf{R}$ and a tuple $t \in R^I$, define $tr(t)$ as the mapping μ such that: (1) the domain of μ is $\{?A \mid A \in att(R)$ and $t.A \neq \texttt{NULL}\}$, and (2) $\mu(?A) = t.A$ for every A in the domain of μ.

Definition 15 (Query Preservation). *A direct mapping $\mathcal{M}$ is* query preserving *if for every relational schema $\mathbf{R}$, set Σ of PKs and FKs over $\mathbf{R}$ and relational algebra query Q over $\mathbf{R}$, there exists a SPARQL query $Q^\star$ such that for every instance I of $\mathbf{R}$ satisfying Σ: $tr([\![Q]\!]_I) = [\![Q^\star]\!]_{\mathcal{M}(\mathbf{R},\Sigma,I)}$.*

We show that the way $\mathcal{DM}$ maps relational data into RDF allows one to answer a query over a relational instance by translating it into an equivalent query over the generated RDF graph.

Theorem 2. *The direct mapping $\mathcal{DM}$ is query preserving.*

Angles and Gutierrez proved that SPARQL has the same expressive power as relational algebra [5]. Thus, one may be tempted to think that this result could be used to prove this theorem. However, the version of relational algebra considered in Angles and Gutierrez does not include the value `NULL` and hence does not apply to $\mathcal{DM}$. The proof is by induction on the structure of a relational query Q. The proof is also constructive and yields a bottom-up algorithm for translating Q into an equivalent SPARQL query.

Before defining monotonicity, consider the following: given two database instances I_1 and I_2 over a relational schema $\mathbf{R}$, instance I_1 is said to be contained in instance I_2, denoted by $I_1 \subseteq I_2$, if for every $R \in \mathbf{R}$, it holds that $R^{I_1} \subseteq R^{I_2}$. A direct mapping $\mathcal{M}$ is considered monotone if for any such pair of instances, the result of mapping I_2 contains the result of mapping I_1. In other words, if we insert new data to the database, then the elements of the mapping that are already computed are unaltered.

Definition 16 (Monotonicity). *A direct mapping $\mathcal{M}$ is monotone if for every relational schema $\mathbf{R}$, set Σ of PKs and FKs over $\mathbf{R}$, and instances I_1, I_2 of $\mathbf{R}$ such that $I_1 \subseteq I_2$: $\mathcal{M}(\mathbf{R}, \Sigma, I_1) \subseteq \mathcal{M}(\mathbf{R}, \Sigma, I_2)$.*

Theorem 3. *The direct mapping $\mathcal{DM}$ is monotone.*

It is straightforward to see that $\mathcal{DM}$ is monotone, because all the negative atoms in the Datalog rules defining $\mathcal{DM}$ refer to the schema, the PKs and the FKs of the database, and these elements are kept fixed when checking monotonicity.

A direct mapping is semantics preserving if the satisfaction of a set of PKs and FKs by a relational database is encoded in the translation process. More precisely, given a relational schema $\mathbf{R}$, a set Σ of PKs and FKs over $\mathbf{R}$ and an instance I of $\mathbf{R}$, a semantics preserving mapping should generate from I a consistent RDF graph if $I \models \Sigma$, and it should generate an inconsistent RDF graph otherwise.

Definition 17 (Semantics Preservation). *A direct mapping $\mathcal{M}$ is semantics preserving if for every relation schema $\mathbf{R}$, set Σ of PKs and FKs over $\mathbf{R}$ and instance I of $\mathbf{R}$: $I \models \Sigma$ iff $\mathcal{M}(\mathbf{R}, \Sigma, I)$ is consistent under OWL semantics.*

Unfortunately, the situation is completely different for the case of semantics preservation, as the following example shows that the direct mapping $\mathcal{DM}$ does not satisfy this property.

Example 6. Assume that a relational schema contains a relation with name `STUDENT` *and attributes* `SID`, `NAME`, *and assume that the attribute* `SID` *is the primary key. Moreover, assume that this relation has two tuples, t_1 and t_2 such that t_1.*`SID` $= 1$, t_1.`NAME` $=$ *John* and t_2.`SID` $= 1$, t_2.`NAME` $=$ *Peter. It is clear that the primary key is violated, therefore the database is inconsistent. However, it is not difficult to see that after applying $\mathcal{DM}$, the resulting RDF graph is consistent.* □

In fact, the result in Example 6 can be generalized as it is possible to show that the direct mapping $\mathcal{DM}$ always generates a consistent RDF graph, hence, it cannot be semantics preserving[7].

Proposition 1. *The direct mapping $\mathcal{DM}$ is not semantics preserving.*

Consider a new direct mapping $\mathcal{DM}_{pk}$ that extends $\mathcal{DM}$ as follows. A Datalog rule is used to determine if the value of a primary key attribute is repeated, and a family of Datalog rules are used to determine if there is a value `NULL` in a column corresponding to a primary key. If some of these violations are found, then an artificial triple is generated that would produce an inconsistency.

If we apply $\mathcal{DM}_{pk}$ to the database of Example 6, it is straightforward to see that starting from an inconsistent relational database, one obtains an RDF graph that is also inconsistent. In fact, we have that:

Proposition 2. *The direct mapping $\mathcal{DM}_{pk}$ is information preserving, query preserving, monotone, and semantics preserving if one considers only PKs. That is, for every relational schema $\mathbf{R}$, set Σ of (only) PKs over $\mathbf{R}$ and instance I of $\mathbf{R}$: $I \models \Sigma$ iff $\mathcal{DM}_{pk}(\mathbf{R}, \Sigma, I)$ is consistent under OWL semantics.*

[7] In practice an RDBMS will not allow a violation of an integrity constraint. However, it may be the case that an RDBMS is not being used and a user may have a dump of data (e.g. in CSV format) and may indicate that a particular column is the primary key when in reality the column violates the constraint.

Information preservation, query preservation and monotonicity of $\mathcal{DM}_{pk}$ are corollaries of the fact that these properties hold for $\mathcal{DM}$, and of the fact that the Datalog rules introduced to handle primary keys are monotone.

The following theorem shows that the desirable condition of being monotone is, unfortunately, an obstacle to obtain a semantics preserving direct mapping.

Theorem 4. *No monotone direct mapping is semantics preserving.*

It is important to understand the reasons why we have not been able to create a semantics preserving direct mapping. The issue is with two characteristics of OWL: (1) it adopts the Open World Assumption (OWA), where a statement cannot be inferred to be false on the basis of failing to prove it, and (2) it does not adopt the Unique Name Assumption (UNA), where two different names can identify the same thing. On the other hand, a relational database adopts the Closed World Assumption (CWA), where a statement is inferred to be false if it is not known to be true. In other words, what causes an inconsistency in a relational database, can cause an inference of new knowledge in OWL.

In order to preserve the semantics of the relational database, we need to ensure that whatever causes an inconsistency in a relational database, is going to cause an inconsistency in OWL. Following this idea, we now present a non-monotone direct mapping, $\mathcal{DM}_{pk+fk}$, which extends $\mathcal{DM}_{pk}$ by introducing rules for verifying beforehand if there is a violation of a foreign key constraint. If such a violation exists, then an artificial RDF triple is created which will generate an inconsistency with respect to the OWL semantics.

It should be noticed that $\mathcal{DM}_{pk+fk}$ is non-monotone because if new data in the database is added which now satisfies the FK constraint, then the artificial RDF triple needs to be retracted.

Theorem 5. *The direct mapping $\mathcal{DM}_{pk+fk}$ is information preserving, query preserving and semantics preserving.*

Information preservation and query preservation of $\mathcal{DM}_{pk+fk}$ are corollaries of the fact that these properties hold for $\mathcal{DM}$ and $\mathcal{DM}_{pk}$.

A direct mapping that satisfies the four properties can be obtained by considering an alternative semantics of OWL that expresses integrity constraints. Because OWL is based on Description Logic, we would need a version of DL that supports integrity constraints, which is not a new idea [17,28,29,34,49,51,72]. Thus, it is possible to extend $\mathcal{DM}_{pk}$ to create an information preserving, query preserving and monotone direct mapping that is also semantics preserving, but it is based on a non-standard version of OWL.

3.3 W3C R2RML: RDB to RDF Mapping Language

R2RML [25] is a language for expressing customized mappings from relational databases to RDF expressed in a graph structure and domain ontology of the user's choice. The R2RML language is also defined as an RDFS schema[8]. An

[8] http://www.w3.org/ns/r2rml.

R2RML mapping is itself represented as an RDF graph. Turtle is the recommended RDF syntax for writing R2RML mappings. The following is an example of an R2RML mapping for the database in Fig. 1. Note that the mapping developer decides which tables and attributes of the database should be exposed as RDF. The Direct Mapping automatically maps all of the tables and attributes of the database.

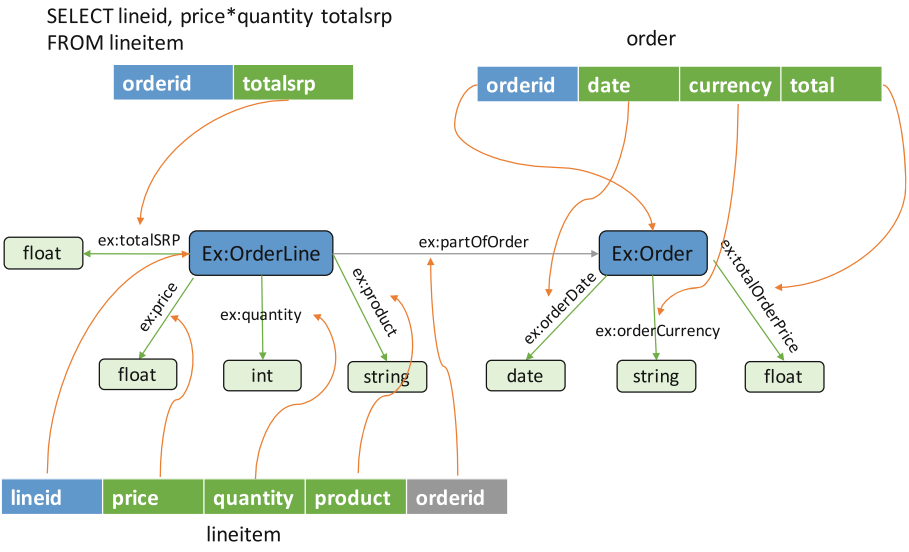


Fig. 3. Example mapping

Example 7 (An R2RML Mapping). Figure 3 represents a mapping from our running example database to an ontology. In this example we will present an R2RML mapping that represents the depiction of Fig. 3.

The target ontology is defined as follows:

```
@prefix ex: <http://ex.com/schema/>.
ex:Order rdf:type owl:Class.
ex:totalOrderPrice rdf:type owl:DatatypeProperty ;
    rdfs:domain ex:Order;
    rdfs:range xsd:float.
ex:orderCurrency rdf:type owl:DatatypeProperty;
    rdfs:domain ex:Order;
    rdfs:range xsd:string.
ex:OrderDate rdf:type owl:DatatypeProperty;
    rdfs:domain ex:Order;
    rdfs:range xsd:date.
```

```
ex:OrderLine rdf:type owl:Class.
ex:price rdf:type owl:DatatypeProperty;
    rdfs:domain ex:OrderLine;
    rdfs:range xsd:float.
ex:quantity rdf:type owl:DatatypeProperty;
    rdfs:domain ex:OrderLine;
    rdfs:range xsd:int.
exproduct rdf:type owl:DatatypeProperty;
    rdfs:domain ex:OrderLine;
    rdfs:range xsd:string.
ex#totalSRP rdf:type owl:DatatypeProperty;
    rdfs:domain ex:OrderLine;
    rdfs:range xsd:float.
ex:partOfOrder rdf:type owl:ObjectProperty;
    rdfs:domain ex:OrderLine;
    rdfs:range ex:Order.
```

The example R2RML Mapping is as follows. In TriplesMap1, all the tuples of the lineitem table are mapped to instances of ex:OrderLine class. The column price, quantity and product of the lineitem table are mapped to the data type properties ex:price ex:quantity and ex:product respectively. The column orderid of the lineitem table which is a foreign key that references orderid of the Order table is mapped to object property ex:partOfOrder. Similarly, in TriplesMap3, all the tuples of the order table are mapped to instances of ex:Order class. The column date, total and currency of the order table are mapped to the data type properties ex:orderDate, ex:totalOrderPrice and ex:orderCurrency respectively. Finally, in TriplesMap2 we have a SQL query that returns a calculation (price*quantity) associated to each lineid. This calculation (the renamed attribute totalsrp) is mapped to the data type property ex:totalSRP.

```
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix ex: <http://ex.com/schema/>.

<#TriplesMap1>
    rr:logicalTable [ rr:tableName "lineitem" ];
    rr:subjectMap [
        rr:template "http://ex.com/data/orderline/{lineid}";
        rr:class ex:OrderLine;
    ];
    rr:predicateObjectMap [
        rr:predicateMap [ rr:constant ex:price ];
        rr:objectMap [ rr:column "price" ];
    ];
    rr:predicateObjectMap [
        rr:predicateMap [ rr:constant ex:quantity ];
        rr:objectMap [ rr:column "quantity" ];
```

```
    ];
    rr:predicateObjectMap [
        rr:predicateMap [ rr:constant ex:product ];
        rr:objectMap [ rr:column "product" ];
    ];
    rr:predicateObjectMap [
        rr:predicate [ rr:constant ex:partOfOrder ];
        rr:objectMap [
            rr:parentTriplesMap <#TriplesMap3>;
            rr:joinCondition [
                rr:child "orderid";
                rr:parent "orderid";
            ];
        ];
    ].

<#TriplesMap2>
    rr:logicalTable [ rr:sqlQuery """
        SELECT lineid, price*quantity totalsrp FROM lineitem
       """ ];
    rr:subjectMap [
        rr:template "http://ex.com/data/orderline/{lineid}";
    ];
   rr:predicateObjectMap [
        rr:predicateMap [ rr:constant ex:totalSRP ];
        rr:objectMap [ rr:column "totalsrp" ];
    ].

<#TriplesMap3>
    rr:logicalTable [ rr:tableName "order" ];
    rr:subjectMap [
        rr:template "http://ex.com/data/order/{orderid}";
        rr:class ex:Order;
    ];
    rr:predicateObjectMap [
        rr:predicateMap [ rr:constant ex:orderDate ];
        rr:objectMap [ rr:column "date" ];
    ];
    rr:predicateObjectMap [
        rr:predicateMap [ rr:constant ex:totalOrderPrice ];
        rr:objectMap [ rr:column "total" ];
    ];
    rr:predicateObjectMap [
        rr:predicateMap [ rr:constant ex:orderCurrency ];
        rr:objectMap [ rr:column "currency" ];
```

```
    ];
 .
```

The following is the resulting RDF after the mapping has been applied on the example database:

```
<http://ex.com/data/order/1234> rdf:type ex:Order;
    ex:orderDate "2017-04-15";
    ex:totalOrderPrice "100";
    ex:orderCurrency "USD".

<http://ex.com/data/orderline/6789> rdf:type ex:OrderLine;
    ex:price "30";
    ex:quantity "2";
    ex:totalSRP "60";
    ex:product "Foo";
    ex:partOfOrder <http://ex.com/data/order/1234>.

<http://ex.com/data/orderline/6790> rdf:type ex:OrderLine
    ex:price "20";
    ex:quantity "2";
    ex:totalSRP "40";
    ex:product "Bar";
    ex:partOfOrder <http://ex.com/data/order/1234>.
```

An R2RML processor may include an R2RML default mapping generator. This is a facility that introspects the schema of the input database and generates an R2RML mapping intended for further customization by a user. This default mapping could be the W3C Direct Mapping or the Direct Mapping as Ontology $\mathcal{DM}$.

The R2RML language features can be divided in two parts: features generating RDF terms (IRI, Blank Nodes or Literals) and features for generating RDF triples.

Generating RDF Terms. An RDF term is either an IRI, a Blank node, or a Literal. A term map generates an RDF term for the subjects, predicates and objects of the RDF triples from either a constant, a template or a column value. A constant-valued term map ignores the row and always generates the same RDF term. A column-valued term map generates an RDF term from the value of a column. A template-valued term map generates an RDF term from a string template, which is a format string that can be used to build strings from multiple components, including the values of a column. Template-valued term maps are commonly used to specify how an IRI should be generated.

The R2RML language allows a user to explicitly state the type of RDF term that needs to be generated (IRI, Blank node or Literal). If the RDF term is for a subject, then the term type must be either an IRI or Blank Node. If the

RDF term is for a predicate, then the term type must be an IRI. If the RDF term is for a subject, then the term type can be either an IRI, Blank node or Literal. Additionally, a developer may assert that an RDF term has an assigned language tag or datatype.

Generating RDF Triples. RDF triples are derived from a logical table. A logical table can be either a base table or view in the relational schema, or an R2RML view. An R2RML view is a logical table whose contents are the result of executing a SQL SELECT query against the input database. In an RDB2RDF mapping, it may be required to transform, compute or filter data before generating RDF triples. This can be achieved by defining a SQL view and referring to it as a base view. However, it may be the case that this is not possible due to lack of sufficient database privileges to create views. R2RML views achieve the same effect without requiring any changes to the input database.

A triples map is the heart of an R2RML mapping. It specifies a rule for translating each row of a logical table to zero or more RDF triples. Example 7 contains two triple maps identified by `<#TriplesMap1>` and `<#TriplesMap2>`. The RDF triples generated from one row in the logical table all share the same subject. A triples map is represented by a resource that references the following other resources:

- It must have exactly one logical table. Its value is a logical table that specifies a SQL query result to be mapped to triples. In Example 7, both Triple Map's 1 and 3 have a table name as a logical table, `lineitem` and `order`, respectively. TripleMap2 has a logical table which is a SQL Query.
- It must have exactly one subject map that specifies how to generate a subject for each row of the logical table.
- It may have zero or more predicate-object maps, which specify pairs of predicate maps and object maps that, together with the subject generated by the subject map, may form one or more RDF triples for each row.

Recall that there are three types of term maps that generate RDF terms: constant-valued, column-valued and template-valued. Given that a subject, predicate and object of an RDF triple must be RDF terms, this means that a subject, predicate and object can be any of the three possible term maps, called subject map, predicate map and object map, respectively. A predicateObject map groups predicate-object map pairs.

A subject map is a term map that specifies the subject of the RDF triple. The primary key of a table is usually the basis for creating an IRI. Therefore, it is normally the case that a subject map is a template-valued term map with an IRI template using the value of a column which is usually the primary key. Consider the triple map `<#TriplesMap1>` in Example 7. The subject map is a template-valued term map where the template is `http://ex.com/data/order/{orderid}`. This means that the subject IRI for each row is formed using values of the `orderid` attribute. Optionally, a subject map may have one or more class IRIs. For each RDF term generated by the

subject map, RDF triples with predicate `rdf:type` and the class IRI as object will be generated. In this example, the class IRI is `ex:Order`.

A predicate-object map is a function that creates one or more predicate-object pairs for each row of a logical table. It is used in conjunction with a subject map to generate RDF triples in a triples map. A predicate-object map is represented by a resource that references the following other resources: One or more predicate maps and one or more object maps or referencing object maps. In `<#TriplesMap1>`, there are four predicate-object maps while `<#TriplesMap2>` only has one.

A predicate map is a term map. It is common that the predicate of an RDF triple is a constant. Therefore, a predicate map is usually a constant-valued term map. For example, the first predicate-object map of `<#TriplesMap1>` has a predicate map which is a constant-valued term map. The predicate IRI will always be the constant is `ex:price`. An object map is also a term map. Several use cases may arise where the object could be either a constant-valued, template-valued or column-valued term map. The first predicate-object map of `<#TriplesMap1>` has an object map which is a column-valued term map. Therefore, the object will be a literal coming from the value of the `price` attribute.

A referencing object map allows using the subjects of another triples map as the objects generated by a predicate-object map. Since both triples maps may be based on different logical tables, this may require a join between the logical tables. A referencing object map is represented by a resource that has exactly one parent triples maps. Additionally, it may have one or more join conditions. Join conditions are represented by a resource that has exactly one value for each of the following: (1) a child, whose value is known as the join condition's child column and must be a column name that exists in the logical table of the triples map that contains the referencing object map (2) a parent, whose value is known as the join condition's parent column and must be a column name that exists in the logical table of the referencing object map's parent triples map. The last predicate-object map of `<#TriplesMap1>` has a referencing object map. The parent triples map is `<#TriplesMap3>`. A join condition is created between the child attribute `orderid`, which is an column name in the logical table of `<#TriplesMap1>` and the parent attribute `orderid`, which is a column name in the logical table of `<#TriplesMap3>`

3.4 Relational Databases to RDF Mappings

Even though there has been attempts to formalize R2RML [62], to the best of our knowledge, there is no formal public definition of R2RML. Nevertheless, we believe it is important to formalize a notion of a customized mapping from Relation Databases to RDF, which we denote as an RDB2RDF mapping. This alternative approach follows the widely used formalization in the data exchange [6] and data integration areas [46], and which is based on the use of first-order logic and its semantics to define mappings.

Given a relational schema $\mathbf{R}$ such that $\texttt{triple} \notin \mathbf{R}$, a class RDB2RDF-rule ρ over $\mathbf{R}$ is a first-order formula of the form:

$$\forall s \forall p \forall o \forall \bar{x}\ \alpha(s, \bar{x}) \wedge p = \texttt{type} \wedge o = c \rightarrow \texttt{triple}(s, p, o), \tag{1}$$

where $\alpha(s, \bar{x})$ is a domain-independent first-order formula over $\mathbf{R}$ and $c \in \mathbf{D}$.

Moreover, a predicate RDB2RDF-rule ρ over $\mathbf{R}$ is a first-order formula of the form:

$$\forall s \forall p \forall o \forall \bar{x}\ \beta(s, o, \bar{x}) \wedge p = c \rightarrow \texttt{triple}(s, p, o), \tag{2}$$

where $\beta(s, o, \bar{x})$ is a domain-independent first-order formula over $\mathbf{R}$ and $c \in \mathbf{D}$. Finally, an RDB2RDF-rule over $\mathbf{R}$ is either a class or a predicate RDB2RDF-rule over $\mathbf{R}$. In what follows, we omit the universal quantifiers $\forall s \forall p \forall o \forall \bar{x}$ from RDB2RDF rules, and we implicitly assume that these variables are universally quantify.

Example 8. Consider the relational database from our running example (see Example 1). Then the following RDB2RDF rule maps all the instances of the `order` *table as instances of the* `Order` *class:* $\texttt{order}(s, x_1, x_2, x_3, x_4, x_5) \wedge p = \texttt{type} \wedge o = \texttt{Order} \rightarrow \texttt{triple}(s, p, o)$.

The RDB2RDF mapping in Example 8 can be represented as follows in R2RML:

```
<#TriplesMap>
    rr:logicalTable [ rr:tableName "order" ];
    rr:subjectMap [
        rr:template "http://ex.com/data/order/{orderid}";
        rr:class ex:Order;
    ];
 .
```

Additionally, it could also be represented as follows:

```
<#TriplesMap>
    rr:logicalTable [ rr:tableName "order" ];
    rr:subjectMap [
        rr:template "http://ex.com/data/order/{orderid}";
    ];
    rr:predicateObjectMap [
        rr:predicate rdf:type ;
        rr:object ex:Order ;
    ];
 .
```

Let $\mathbf{R}$ be a relational schema. An RDB2RDF mapping $\mathcal{M}$ over $\mathbf{R}$ is a finite set of RDB2RDF rules over $\mathbf{R}$. Given an RDB2RDF mapping $\mathcal{M}$ and an instance

I over $\mathbf{R}$, the result of applying $\mathcal{M}$ over I, denoted by $[\![\mathcal{M}]\!]_I$, is an instance over the schema $\{\texttt{triple}\}$ that is defined as the result of the following process. For every RDB2RDF rule of the form (1) and value $c_1 \in \mathbf{D}$, if there exists a tuple of values $\bar{d}$ from $\mathbf{D}$ such that $I \models \alpha(c_1, \bar{d})$,[9] then $\texttt{triple}(c_1, \texttt{type}, c)$ is included as a fact of $[\![\mathcal{M}]\!]_I$, and likewise for every RDB2RDF rule of the form (2). Notice that this definition coincides with the notion of canonical universal solution in the context of data exchange [6]. Besides, notice that $[\![\mathcal{M}]\!]_I$ represents an RDF graph and, thus, mapping $\mathcal{M}$ can be considered as a mapping from relational databases into RDF graphs.

Example 9. Consider the relational database from our running example, and let $\mathcal{M}$ be an RDB2RDF mapping consisting of the rule in Example 1 and the following rule:

$$\texttt{order}(s, x_1, o, x_3, x_4, x_5) \wedge p = \texttt{orderDate} \rightarrow \texttt{triple}(s, p, o) \tag{3}$$

If I is the instance from our running example, then $[\![\mathcal{M}]\!]_I$ consists of the following facts:

$$\texttt{triple}(1234, \texttt{type}, \texttt{Order}),\ \texttt{triple}(1234, \texttt{orderDate}, 2017-04-15).$$

The RDB2RDF mapping in Example 9 can be represented as follows in R2RML:

```
<#TriplesMap>
    rr:logicalTable [ rr:tableName "order" ];
    rr:subjectMap [
        rr:template "http://ex.com/data/order/{orderid}";
        rr:class ex:Order;
    ];
    rr:predicateObjectMap [
        rr:predicate  ex:orderDate ;
        rr:objectMap [ rr:column "date" ];
    ];
.
```

4 From the Semantic Web to Relational Databases: Data Access

The Semantic Web's promise of web-wide data integration requires the inclusion of legacy Relational Databases. In the previous section, we discussed how to go from a Relational Database to the Semantic Web through means of mappings. In this section, we present the other direction: how the Semantic Web can access a Relational Database.

[9] Given that $\alpha(s, \bar{x})$ is domain-independent, there exists a finite number of tuples $(c_1, \bar{d})$ such that $I \models \alpha(c_1, \bar{d})$.

In RDF data management there are efforts that concern Triplestores and those that concern legacy Relational Databases. Triplestores are database management systems whose data model is RDF, and support at least SPARQL execution against the stored contents. Native triplestores are those that are implemented from scratch [14,53,76]. RDBMS-backed Triplestores are built by adding an application layer to an existing relational database management system. Within that literature there is a discourse concerning the best database schema, SPARQL to SQL query translations, indexing methods and even storage managers, (i.e. column stores vs. row stores) [1,21,30,77]. NoSQL Triplestores are also being investigated as possible RDF storage managers [24,31,41,44]. In all three triplestore cases (native, RDBMS-backed and NoSQL), RDF is the primary data model.

The research herein is concerned with the mapping of legacy relational data with the Semantic Web. Within that, the research concerns wrapper systems that present a logical RDF representation of relational data that is physically stored in an RDBMS such that no copy of the relational data is made. It follows that some or all of a SPARQL query evaluation is executed by the SQL engine. An alternative approach is the one in which the relational data is extracted from the relational database, transformed to RDF, and loaded (ETL) into a Triplestore.

Since both RDBMS-backed Triplestores and RDB2RDF Wrapper systems involve relational databases and translation from SPARQL to SQL, there is a potential for confusion. The difference is that RDBMS-backed Triplestores translate SPARQL queries to SQL queries that are executed on database schemas that model and store RDF. RDB2RDF Wrapper systems translate SPARQL queries to SQL queries that are executed on legacy database schemas that model and store relational data.

An RDB2RDF ETL approach is recommended when the data in the legacy relational database is stale, or updated infrequently. In an ETL system, at best, updates occur on a regular cycle. Thus semantic web applications querying stale data just prior to an update is a risk. In the common case of legacy relational databases which are continually updated, an ETL approach is not feasible. A solution to this problem is the use of a RDB2RDF wrapper systems which compiles SPARQL to SQL.

4.1 SPARQL to SQL Rewriting with Direct Mapping

In mid to late 2000s, RDB2RDF wrapper systems such as D2RQ, Virtuoso RDF Views and Squirrel RDF, predicated on preprocessing and/or optimizing the SQL query before sending it to the SQL optimizer. Open-source code and forums[10] provide evidence of their architecture. For example, we observed that for some SPARQL queries, D2RQ generates multiple SQL queries and necessarily executed a join among those results outside of the database. In 2011, we postulated that by carefully constructing SQL views to represent a RDB2RDF

[10] https://github.com/d2rq/d2rq/issues/94 As of April 2017, this issue is still open.

mapping, then the existing algorithmic machinery in SQL optimizers were sufficient to effectively execute SPARQL queries on native relational data [67]. Thereby, legacy relational database systems may be made upwardly compatible with the Semantic Web, while simultaneously minimizing the complexity of the wrapping system.

In 2008, Angles and Gutierrez showed that SPARQL is equivalent in expressive power to relational algebra [5]. Thus, one might have expected that the validity of this research's postulate at that time, to be a foregone conclusion. However, in 2009, two independent studies that evaluated three RDB2RDF wrapper systems, D2RQ, Virtuoso RDF Views and Squirrel RDF, came to the opposite conclusion: existing SPARQL to SQL translation systems do not compete with traditional relational databases [11,33].

The March 2009 Berlin SPARQL Benchmark on the 100 million triple dataset reported that SPARQL queries on the evaluated RDB2RDF systems were up to 1000 times slower that the native SQL queries. Bizer and Schultz [11], creators of the Berlin SPARQL Benchmark, concluded that: "*Setting the results of the RDF stores and the SPARQL-to-SQL rewriters in relation to the performance of classical RDBMS unveiled an unedifying picture. Comparing the overall performance (100 M triple, single client, all queries) of the fastest rewriter with the fastest relational database shows an overhead for query rewriting of 106%. This is an indicator that there is still room for improving the rewriting algorithms*".

Gray et al. [33] tested D2RQ and SquirrelRDF on a scientific database. This study concluded that "*... current rdb2rdf systems are not capable of providing the query execution performance required to implement a scientific data integration system based on the rdf model. [...] it is likely that with more work on query translation, suitable mechanisms for translating queries could be developed. These mechanisms should focus on exploiting the underlying database system's capabilities to optimize queries and process large quantities of structured data, e.g. pushing the selection conditions to the underlying database system*".

A motivation for this research, at that time, was to resolve the apparent contradiction among the aforementioned papers. Toward that end we researched and engineered the Ultrawrap system [67].

Ultrawrap Architecture. The first version of Ultrawrap was compliant with the W3C Direct Mapping standard. The goal was to understand if existing commercial relational databases already subsume the algorithms and optimizations needed to support effective SPARQL execution on existing relationally stored data under the simplest mapping possible. This initial version of Ultrawrap was organized as a set of four compilers with the understanding that the SQL optimizer formed one of the compilers.

1. The generation of the Direct Mapping with the translation of a SQL schema, including constraints, to an OWL ontology: the putative ontology (PO).
2. The creation of an intensional triple table in the database by augmenting the relational schema with one or more SQL Views: the Tripleview.

3. Translation of SPARQL queries to equivalent SQL queries operating on the Tripleview.
4. The native SQL query optimizer, which becomes responsible for rewriting triple based queries and effecting their execution on extensional relational data.

These four components can be seen as four different language compilers. As an ensemble, the first three provide for the logical mapping of schema, data and queries between the relational and Semantic Web languages. The fourth component, the SQL optimizer, is responsible for the evaluation of the data mappings and concomitant optimization of the query.

To define the mapping of the relational data to RDF, the system first identifies an ontological representation of the relational schema, which is done by the Direct Mapping and the generation of the putative ontology. The putative ontology is the input to a second compilation step that creates a logical definition of the relational data as RDF and embeds it in a view definition. In a off-line process, Ultrawrap defines a SQL view whose query component is a specification of a mapping from the relational data to an RDF triple representation, the Tripleview. Per the Direct Mapping, concatenating the table name with the primary key value or table name with attribute name creates unique identifiers for subject, predicate and objects. Subsequently, unique identifiers can be appended to a base URI. The SQL Tripleview is comprised of a union of SELECT-FROM-WHERE (SFW) statements. The WHERE clause filters attributes with null values (IS NOT NULL), given that null values are not expressible in RDF.

Due to its simplicity, our starting point is the triple table approach. Even though, studies have shown that storing RDF with the triple table approach in a relational database is easily improved upon [1,48], this issue is not relevant to Ultrawrap because the relational data is not being materialized in a triple table; instead the relational data is virtually represented as a triple table through unmaterialized views.

Even though our goal is to define a virtual triple table, we still have to anticipate the physical characteristics of the database and the capacity of the SQL optimizer to produce optimal physical plans. Toward that end, the Tripleview has the following characteristics.

The Tripleview is of the form: <subject, primary key of subject, predicate, object, primary key of object>. Separating the primary key in the Tripleview allows the query optimizer to exploit them because the joins are done on these values. If the object is a data value, then a NULL is used as the primary key of the object. The subject and object are still kept as the concatenation of the table name with the primary key value because this is used to generate the final URI, which uniquely identifies each tuple in the database. It is possible to augment the number of attributes in the Tripleview to include each separate key value.

Instead of having a single Tripleview to represent the entire mapping, it is beneficial to create a separate Tripleview for each datatype. For varchar, this includes each length declared in the schema. For example, datatypes with varchar(50) and varchar(200) are considered different. Using multiple Tripleviews

requires less bookkeeping than one might anticipate. Each attribute is mapped to its corresponding Tripleview and stored in a hashtable. Then, given an attribute, the corresponding Tripleview can be retrieved.

For example, the Tripleviews for the direct mapping of our running example is the following:

```
CREATE VIEW Tripleview_type(s,s_id,p,o,o_id) AS
SELECT "order"+orderid as s, orderid as s_id, "type" as p,
       "order" as o, null as o_id
FROM order
UNION ALL
SELECT "lineitem"+lineid as s, lineid as s_id,"type" as p,
       "lineitem" as o, null as o_id
FROM lineitem

CREATE VIEW Tripleview_int(s,s_id,p,o,o_id) AS
SELECT "order"+orderid as s, orderid as s_id, "orderid" as p,
       orderid as o, null as o_id
FROM order WHERE orderid IS NOT NULL
UNION ALL
SELECT "order"+orderid as s, orderid as s_id, "status" as p,
      status as o, null as o_id
FROM order WHERE status IS NOT NULL
UNION ALL
SELECT "lineitem"+lineid as s, lineid as s_id,"price" as p,
     price as o, null as o_id
FROM lineitem WHERE price IS NOT NULL
UNION ALL
SELECT "lineitem"+lineid as s, lineid as s_id,"quantity" as p,
      quantity as o, null as o_id
FROM lineitem WHERE quantity IS NOT NULL
UNION ALL
SELECT "lineitem"+lineid as s, lineid as s_id,"orderid" as p,
      orderid as o, null as o_id
FROM lineitem WHERE orderid IS NOT NULL

CREATE VIEW Tripleview_varchar50(s,s_id,p,o,o_id) AS
SELECT "order"+orderid as s, orderid as s_id, "currency" as p,
      currency as o, null as o_id
FROM order WHERE currency IS NOT NULL
UNION ALL
SELECT "lineitem"+lineid as s, lineid as s_id,"product" as p,
      product as o, null as o_id
FROM lineitem WHERE product IS NOT NULL

CREATE VIEW Tripleview_float(s,s_id,p,o,o_id) AS
SELECT "order"+orderid as s, orderid as s_id, "total" as p,
       total as o, null as o_id
FROM order WHERE total IS NOT NULL
UNION ALL
```

```
SELECT "lineitem"+lineid as s, lineid as s_id,"price" as p,
       price as o, null as o_id
FROM lineitem WHERE price IS NOT NULL

CREATE VIEW Tripleview_object(s,s_id,p,o,o_id) AS
SELECT "lineitem"+lineid as s, lineid as s_id,
       "lineitem#ref-orderid" as p, "order"+orderid as o, orderid as o_id
FROM lineitem WHERE orderid IS NOT NULL
```

Ultrawrap's runtime phase encompasses the translation of SPARQL queries to SQL queries on the Tripleviews and the maximal use of the SQL infrastructure to do the SPARQL query rewriting and execution. At runtime, a compiler translates an incoming SPARQL query to a SQL query in terms of the Tripleview. The translation of the SPARQL query to a SQL query on the Tripleviews follows a classic compiler structure: a parser converts the SPARQL query string to an Abstract Syntax Tree (AST). The AST is translated into an SPARQL algebra expression tree. The SQL translation is accomplished by traversing the expression tree and replacing each SPARQL operator. Each internal node of the expression tree represents a SPARQL binary algebra operator while the leaves represent a Basic Graph Patterns (BGP), which is a set of triple patterns. A SPARQL BGP is a set of triple patterns where each one maps to a Tripleview. A SPARQL Join maps to a SQL Inner Join, a SPARQL Union maps to the SQL Union, a SPARQL Optional maps to SQL Left-Outer Join. Consequently, the RDBMS must use both the logical mapping represented in the Tripleview and optimize the resulting translated SQL query, forming the final compiler.

Example 10. The following SPARQL query returns all the quantity and products in a line item.

```
SELECT ?quantity ?product
WHERE {
 ?x <lineitem#quantity> ?quantity.
 ?x <lineitem#product>  ?product.
}
```

The Ultrawrap SQL query is the following:

```
SELECT t1.o AS quantity, t2.o AS product
FROM Tripleview_varchar50 t1, Tripleview_int t2
WHERE t1.p = "quantity"AND t2.p ="product"
AND t1.s = t2.s AND t1.s_id = t2.s_id
```

Two Important Optimizations. Upon succeeding in ultrawrapping different RDBMSs and reviewing query plans, two relational optimizations emerged as important for effective execution of SPARQL queries: (1) detection of unsatisfiable conditions and (2) self-join elimination. Perhaps, not by coincidence, these two optimizations are among semantic query optimization (SQO) methods introduced in the 1980's [18,20,70]. In SQO, the objective is to leverage

the semantics, represented in integrity constraints, for query optimization. The basic idea is to use integrity constraints to rewrite a query into a semantically equivalent one. These techniques were initially designed for deductive databases and then integrated in commercial relational databases [20].

The idea behind the **detection of unsatisfiable conditions optimization** is to determine that a query result is empty by determining, without executing the query. This happens, for example, when a pair of predicate constants are inconsistent [18]. The application of the following transformations eliminates columns from the plan that are not needed to evaluate the SPARQL query.

Elimination by contradiction: Consider a query `SELECT * FROM R WHERE A=x AND A=y such that x != y`. Then the result of that query is empty. For example, it is clear that the query `SELECT * FROM order WHERE orderid = 1 AND orderid = 2` will never return results.

Unnecessary union sub-tree pruning: Given a query that includes the UNION operator and where it has been determined that an argument of the UNION is empty; then the corresponding argument can be eliminated. For example: `UNION ALL ({}, S, T) = UNION ALL (S, T)` and `UNION ALL ({}, T) = T`

In Ultrawrap's Tripleview, the constant value in the predicate position acts as the integrity constraint. Consider the following Tripleview:

```
CREATE VIEW Tripleview_varchar50(s,s_id,p,o,o_id) AS
SELECT "order"+orderid as s, orderid as s_id, "currency" as p,
        currency as o, null as o_id FROM order
WHERE currency IS NOT NULL
UNION ALL
SELECT "lineitem"+lineid as s, lineid as s_id,"product" as p,
        product as o, null as o_id FROM lineitem
WHERE product IS NOT NULL
```

Now consider the following query "return all product labels":

```
SELECT o FROM Tripleview_varchar50 WHERE p = "product"
```

The first SFW statement from Tripleview_varchar50 defines `p="currency"`. The query contains `p ="product"`. Both predicates cannot be satisfied simultaneously. Given the contradiction, the first SFW statement of Tripleview_varchar50 can be replaced with the empty set. Since the Tripleview's definition includes all possible columns, any specific SPARQL query will only need a subset of the statements defined in the view. Application of elimination by contradiction enables removing, the unnecessary UNION ALL conditions. Thus the combination of the two transformations reduces the Tripleview to precisely the subset of referenced columns.

Example 11. Consider the Ultrawrap SQL query in Example 10, after applying the detection of unsatisfiable condition optimization, the new Ultrawrap SQL query would *logically* be the following

```
SELECT t1.o AS quantity, t2.o AS product
FROM
    (SELECT"lineitem"+lineid as s, lineid as s_id,"quantity"as p,
     quantity as o, null as o_id FROM lineitem WHERE quantity IS NOT NULL) t1,
    (SELECT"lineitem"+lineid as s, lineid as s_id,"product"as p,
     product as o, null as o_id FROM lineitem WHERE product IS NOT NULL) t2
    WHERE t1.p ="quantity"AND t2.p ="product"
    AND t1.s = t2.s AND t1.s_id = t2.s_id
```

Join elimination is one of the several SQO techniques, where integrity constraints are used to eliminate a literal clause in the query. This implies that a join could also be eliminated if the table that is being dropped does not contribute any attributes in the results [18]. The type of join elimination that is desired is the **self-join elimination**, where a join occurs between the same tables. Two different cases are observed: self-join elimination of projection and self-join elimination of selections.

Self-join elimination of projection: This occurs when attributes from the same table are projected individually and then joined together. For example, the following unoptimized query projects the attributes total and currency from the table order where orderid = 1, however each attribute projection is done separately and then joined:

```
SELECT p1.total, p2.currency
FROM order p1, order p2
WHERE p1.orderid = 1 AND p1.orderid = p2.orderid
```

Given a self-join elimination optimization, the previous query may be rewritten as:

```
SELECT total, currency FROM order WHERE orderid = 1
```

Self-join elimination of selection: This occurs when a selection on attributes from the same table are done individually and then joined together. For example, the following unoptimized query selects on price > 100 and quantity > 10 separately and then joined:

```
SELECT p1.lineid
FROM lineitem p1, lineitem p2
WHERE p1.price > 100 AND p2.quantity > 10 AND p1.lineid = p2.lineid
```

Given a self-join elimination optimization, the previous query may be rewritten as:

```
SELECT lineid FROM lineitem WHERE price > 100 AND quantity > 10
```

Example 12. Consider the logical Ultrawrap SQL query in Example 11. After the self join elimination optimization has been applied, the new Ultrawrap SQL query would *logically* be the following

```
SELECT t1.quantity, t1.product
FROM lineitem t1
WHERE t1.quantity IS NOT NULL and t1.product IS NOT NULL
```

Evaluation. Ultrawrap was evaluated using the three leading RDBMS systems and two benchmark suites, Microsoft SQL Server, IBM DB2 and Oracle RDBMS, and the Berlin and Barton SPARQL benchmarks. The SPARQL benchmarks were chosen as a consequence of the fact that they derived their RDF content from a relational source. Both benchmark provide both SPARQL queries and SQL queries, where each query was derived independently from an English language specification. Since wrappers produce SQL from SPARQL we refer to the benchmark's SQL queries as benchmark-provided SQL queries.

By using benchmarks containing independently created SPARQL and SQL queries, and considering the effort and maturity embodied in the leading RDBMS's SQL optimizers, we suppose that the respective benchmark-provided SQL query execution time forms a worthy baseline, and the specific query plans to yield insight into methods for creating wrappers.

By starting with a simple wrapper system and evaluating it with sophisticated SQL query optimizers we are able to identify existing, well understood optimization methods that enable wrappers. We determined that DB2 implements both optimizations. SQL Server implements the detection of unsatisfiable conditions optimization but does not implement the self-join elimination optimization. Oracle does not implement the detection of unsatisfiable conditions optimization. It does implement the self-join elimination optimization, but only if the detection of unsatisfiable conditions optimization is applied separately. MySQL does not implement any of these optimizations.

The following points deserve elaboration:

- Self-join elimination: The number of self-joins and their elimination is not, by itself, an indicator of poor performance. The impact of the self-join elimination optimization is a function of the selectivity and the number of properties in the SPARQL query that are co-located in a single table. The value of optimization is less as selectivity increases.
- Join predicate push-down: The experiments with Oracle revealed that pushing join predicates [3] can be as effective as the detection of unsatisfiable conditions optimization.
- Join ordering: Join order is a major factor for poor query execution time, both on Ultrawrap and benchmark-provided SQL queries.
- Left-outer joins: We found that no commercial optimizer eliminates self left-outer joins and OPTIONALs appear in many of the queries where sub-optimal join orders are determined. We speculate that these types of queries are not common in a relational setting, hence the lack of support in commercial systems.
- Counting NULLs: Each SFW statement of the Tripleview filters null values. Such a filter could produce an overhead, however we speculate that the optimizer has statistics of null values and avoids the overhead.

The results of the Ultrawrap system provided a foundation for identifying minimal requirements for effective SPARQL to SQL wrapper systems. Since then, other research groups have continued this work and developed systems such as Morph [58] and Ontop [61].

4.2 Ontology-Based Data Access

In the previous section, we presented the initial Ultrawrap system, who focus is on supporting a Direct Mapping. In this section, we present how Ultrawrap has been extended for Ontology-Based Data Access, denoted as Ultrawrap$^{\text{OBDA}}$, and thus supports customized mappings in R2RML [66].

Given a source relational database, a target OWL ontology and a mapping from the relational database to the ontology, Ontology-Based Data Access (OBDA) concerns answering queries over the target ontology using these three components. Commonly, researchers have taken two approaches to developing OBDA systems: materialization-based approach (forward chaining) or rewriting-based approach (backward chaining). In the materialization approach, the input relational database D, target ontology $\mathcal{O}$ and mapping $\mathcal{M}$ (from D to $\mathcal{O}$) are used to derive new facts that are stored in a database D_o, which is considered to be the materialization of the data in D given $\mathcal{M}$ and $\mathcal{O}$. Then the answer to a SPARQL query Q over the target ontology over D, $\mathcal{M}$ and $\mathcal{O}$ is computed by directly posing Q over D_o [6]. In the rewriting approach, three steps are executed. First, a new query Q_o is generated from the query Q and the ontology $\mathcal{O}$, which is considered to be the rewriting of Q w.r.t. to $\mathcal{O}$. The majority of the OBDA literature focuses on this step [54]. Second, the mapping $\mathcal{M}$ is used to compile Q_o to a SQL query Q_{sql} over D [56,57]. Finally, Q_{sql} is evaluated on the database D, which gives us the answer to the initial query Q. Therefore, the answer to a query Q over O, D, and M is computed by directly posing Q_{sql} over D.

We develop an OBDA system, Ultrawrap$^{\text{OBDA}}$, which combines materialization and query rewriting. Ultrawrap$^{\text{OBDA}}$ is an extension of our previous Ultrawrap system which supports customized mappings in R2RML. In the same spirit of our Ultrawrap work, the objective is to effect optimizations by pushing processing into the Relational Databases Management Systems (RDBMS) and closer to the stored data, hence making maximal use of existing SQL infrastructure. We distinguish two phases: a compile and runtime phase. In the compile phase, we are given as input a relational database D, an ontology $\mathcal{O}$ and a mapping $\mathcal{M}$ from D to $\mathcal{O}$. The mapping $\mathcal{M}$ is given in R2RML. The first step of this phase is to embed in $\mathcal{M}$ the ontological entailments of $\mathcal{O}$, which gives rise to a new mapping $\mathcal{M}^{\star}$, that is called the *saturation* of $\mathcal{M}$ w.r.t. $\mathcal{O}$. The mapping $\mathcal{M}^{\star}$ is implemented using SQL views. In order to improve query performance, an important issue is to decide which views should be materialized. This is the last step of the compilation phase. We then study when a view should be materialized in order to improve query performance. In the runtime phase, the input is a query Q over the target ontology $\mathcal{O}$, which is written in the RDF query language SPARQL, and the problem is to answer this query by rewriting it into some SQL queries over the views. A key observation at this point is that some existing SQL optimizers are able to perform rewritings in order to execute queries against materialized views.

To the best of our knowledge, in 2014, we presented the first OBDA system which supported ontologies with transitivity by using SQL recursion. The ontology profile considered in this work is our proposed OWL-SQL. More

specifically, our contributions are the following. (1) We present an efficient algorithm to generate saturated mappings. (2) We provide a proof that every SPARQL query over a target ontology can be rewritten into a SQL query in our context, where mappings play a fundamental role. It is important to mention that such a result is a minimal requirement for a query-rewriting OBDA system relying on relational database technology. (3) We present a cost model that help us to determine which views to materialize to attain the fastest execution time. And (4) we present an empirical evaluation using (i) Oracle, (ii) two benchmarks including an extension of the Berlin SPARQL Benchmark, and (iii) six different scenarios. This evaluation includes a comparison against a state-of-the-art OBDA system, and its results validate the cost model and demonstrate favorable execution times for Ultrawrap$^{\text{OBDA}}$.

Related work. This research builds upon the work of Rodriguez-Muro et al. implemented in Ontop [61] and our previous work on Ultrawrap [67]. Rodriguez-Muro et al. uses the tree-witness rewriting algorithm and introduced the idea of compiling ontological entailments as mappings, which they named $\mathcal{T}$-Mappings. There are three key differences between Rodriguez-Muro et al. and our work in this paper: (1) we have extended the work of Rodriguez-Muro et al. to support more than hierarchy of classes and properties, including transitivity; (2) we introduce an efficient algorithm that generates saturated mappings while Rodriguez-Muro et al. has not presented an algorithm before; and (3) we represent the mappings as SQL views and study when the views should be materialized. Ultrawrap is a system that encodes a fix mapping, the direct mapping [7,65], of the database as RDF. These mappings are implemented using unmaterialized SQL views. The approach presented extends Ultrawrap in three important aspects: (1) supports a customized mapping language; (2) supports reasoning through saturated mappings; and (3) considers materializing views for query optimization. Another related work is the combined approach [47], which materializes entailments as data, without considering mappings, and uses a limited form of query rewriting. The main objective of this approach is to deal with the case of infinite materialization, which cannot occur for the type of ontologies considered in this paper.

Saturation of RDB2RDF Mappings. Being able to modify an RDB2RDF mapping to embed a given ontology is a fundamental step in our approach. This process is formalized by means of the notion of saturated mapping.

Definition 18 (Saturated Mapping). *Let $\mathcal{M}$ and $\mathcal{M}^\star$ be RDB2RDF mappings over a relational schema $\mathbf{R}$ and $\mathcal{O}$ an ontology. Then $\mathcal{M}^\star$ is a saturation of $\mathcal{M}$ w.r.t. $\mathcal{O}$ if for every instance I over $\mathbf{R}$ and assertional RDF-triple (a, b, c):*

$$[\![\mathcal{M}]\!]_I \cup \Sigma_{\mathcal{O}} \models \texttt{triple}(a, b, c) \quad \textit{iff} \quad \texttt{triple}(a, b, c) \in [\![\mathcal{M}^\star]\!]_I.$$

We study the problem of computing a saturated mapping from a given mapping and ontology. In particular, we focus on the case of ontologies not mentioning any triple of the form $(a, \texttt{type}, \texttt{transProp})$, which we denote by

non-transitive ontologies. In the next section, we extend these results to the case of arbitrary ontologies.

In our system, the saturation step is performed by exhaustively applying the inference rules in Table 1, which allow us to infer new RDB2RDF rules from the existing ones and the input ontology. More precisely, given an inference rule $t{:}\frac{\rho_1}{\rho_2}$ from Table 1, where t is a triple and ρ_1, ρ_2 are RDB2RDF rules, and given an RDB2RDF mapping $\mathcal{M}$ and an ontology $\mathcal{O}$, we need to do the following to apply $t{:}\frac{\rho_1}{\rho_2}$ over $\mathcal{M}$ and $\mathcal{O}$. First, we have to replace the letters $\mathtt{A}$ and $\mathtt{B}$ in t with actual URIs, say $a \in \mathbf{I}$ and $b \in \mathbf{I}$, respectively.[11] Second, we need to check whether the triple obtained from t by replacing $\mathtt{A}$ by a and $\mathtt{B}$ by b belongs to $\mathcal{O}$, and whether the RDB2RDF rule obtained from ρ_1 by replacing $\mathtt{A}$ by a belongs to $\mathcal{M}$. If both conditions hold, then the inference rule can be applied, and the result is an RDB2RDF mapping $\mathcal{M}'$ consisting of the rules in $\mathcal{M}$ and the rule obtained from ρ_2 by replacing $\mathtt{A}$ by a and $\mathtt{B}$ by b.

Table 1. Inference rules to compute saturated mappings.

$$(\mathtt{A}, \mathtt{subClass}, \mathtt{B}) : \frac{\alpha(s,\bar{x}) \wedge p = \mathtt{type} \wedge o = \mathtt{A} \rightarrow \mathtt{triple}(s,p,o)}{\alpha(s,\bar{x}) \wedge p = \mathtt{type} \wedge o = \mathtt{B} \rightarrow \mathtt{triple}(s,p,o)}$$

$$(\mathtt{A}, \mathtt{subProp}, \mathtt{B}) : \frac{\beta(s,o,\bar{x}) \wedge p = \mathtt{A} \rightarrow \mathtt{triple}(s,p,o)}{\beta(s,o,\bar{x}) \wedge p = \mathtt{B} \rightarrow \mathtt{triple}(s,p,o)}$$

$$(\mathtt{A}, \mathtt{dom}, \mathtt{B}) : \frac{\beta(s,o,\bar{x}) \wedge p = \mathtt{A} \rightarrow \mathtt{triple}(s,p,o)}{\beta(s,y,\bar{x}) \wedge p = \mathtt{type} \wedge o = \mathtt{B} \rightarrow \mathtt{triple}(s,p,o)}$$

$$(\mathtt{A}, \mathtt{range}, \mathtt{B}) : \frac{\beta(s,o,\bar{x}) \wedge p = \mathtt{A} \rightarrow \mathtt{triple}(s,p,o)}{\beta(y,s,\bar{x}) \wedge p = \mathtt{type} \wedge o = \mathtt{B} \rightarrow \mathtt{triple}(s,p,o)}$$

$$\begin{array}{r}(\mathtt{A}, \mathtt{equivClass}, \mathtt{B}) \\ \text{or } (\mathtt{B}, \mathtt{equivClass}, \mathtt{A})\end{array} : \frac{\alpha(s,\bar{x}) \wedge p = \mathtt{type} \wedge o = \mathtt{A} \rightarrow \mathtt{triple}(s,p,o)}{\alpha(s,\bar{x}) \wedge p = \mathtt{type} \wedge o = \mathtt{B} \rightarrow \mathtt{triple}(s,p,o)}$$

$$\begin{array}{r}(\mathtt{A}, \mathtt{equivProp}, \mathtt{B}) \\ \text{or } (\mathtt{B}, \mathtt{equivProp}, \mathtt{A})\end{array} : \frac{\beta(s,o,\bar{x}) \wedge p = \mathtt{A} \rightarrow \mathtt{triple}(s,p,o)}{\beta(s,o,\bar{x}) \wedge p = \mathtt{B} \rightarrow \mathtt{triple}(s,p,o)}$$

$$\begin{array}{r}(\mathtt{A}, \mathtt{inverse}, \mathtt{B}) \\ \text{or } (\mathtt{B}, \mathtt{inverse}, \mathtt{A})\end{array} : \frac{\beta(s,o,\bar{x}) \wedge p = \mathtt{A} \rightarrow \mathtt{triple}(s,p,o)}{\beta(o,s,\bar{x}) \wedge p = \mathtt{B} \rightarrow \mathtt{triple}(s,p,o)}$$

$$(\mathtt{A}, \mathtt{type}, \mathtt{symProp}) : \frac{\beta(s,o,\bar{x}) \wedge p = \mathtt{A} \rightarrow \mathtt{triple}(s,p,o)}{\beta(o,s,\bar{x}) \wedge p = \mathtt{A} \rightarrow \mathtt{triple}(s,p,o)}$$

Example 13. Consider the RDB2RDF rule $\mathtt{order}(s, x_1, x_2, x_3, x_4, \mathtt{1}) \wedge p = \mathtt{type} \wedge o = \mathtt{ShippedOrder} \rightarrow \mathtt{triple}(s,p,o)$., *and assume that we are given an ontology* $\mathcal{O}$ containing the triple $(\mathtt{ShippedOrder}, \mathtt{subClass}, \mathtt{SuccessfulOrder})$. *Then by applying the first inference rule in Table* 1, *we infer the following RDB2RDF rule:* $\mathtt{order}(s, x_1, x_2, x_3, x_4, \mathtt{1}) \wedge p = \mathtt{type} \wedge o = \mathtt{SuccessfulOrder} \rightarrow \mathtt{triple}(s,p,o)$.

[11] If $t = (\mathtt{A}, \mathtt{type}, \mathtt{symProp})$, then we only need to replace $\mathtt{A}$ by a.

Given an RDB2RDF mapping $\mathcal{M}$ and an ontology $\mathcal{O}$, we denote by $\text{SAT}(\mathcal{M}, \mathcal{O})$ the RDB2RDF mapping obtained from $\mathcal{M}$ and $\mathcal{O}$ by successively applying the inference rules in Table 1 until the mapping does not change. The following theorem shows that $\text{SAT}(\mathcal{M}, \mathcal{O})$ is a saturation of $\mathcal{M}$ w.r.t. $\mathcal{O}$, which justifies its use in our system.

Theorem 6. *For every RDB2RDF mapping $\mathcal{M}$ and ontology $\mathcal{O}$ in RDFS, it holds that $\text{SAT}(\mathcal{M}, \mathcal{O})$ is a saturation of $\mathcal{M}$ w.r.t. $\mathcal{O}$.*

Theorem 6 is a corollary of the fact that the first six rules in Table 1 encode the rules to infer assertional triples from an inference system for RDFS given in [52].

A natural question at this point is whether $\text{SAT}(\mathcal{M}, \mathcal{O})$ can be computed efficiently. In our setting, the approach based on exhaustively applying the inference rules in Table 1 can be easily transformed into a polynomial time algorithm for this problem. However, if this transformation is done in a naïve way, then the resulting algorithm is not really efficient. In [66], we present an efficient algorithm to compute $\text{SAT}(\mathcal{M}, \mathcal{O})$ that is linear in the size of the input RDB2RDF mapping $\mathcal{M}$ and ontology $\mathcal{O}$, which are denoted by $\|\mathcal{M}\|$ and $\|\mathcal{O}\|$, respectively.

Theorem 7. *There exists an algorithm that, given an RDB2RDF mapping $\mathcal{M}$ and a non-transitive ontology $\mathcal{O}$, computes $\text{SAT}(\mathcal{M}, \mathcal{O})$ in time $O(\|\mathcal{M}\| \cdot \|\mathcal{O}\|)$.*

The main ingredients of the algorithm mentioned in Theorem 7 can be found in [66].

Dealing with Transitive Predicates. We show here how the approach presented in the previous section can be extended with recursive predicates. This functionality is of particular interest as the current work on OBDA under OWL 2 QL does not consider transitivity, mainly because the query language in which the query over the ontology has to be rewritten is SQL without recursion [15].

From now on, given a first-order formula $\varphi(x, y)$, we use $\text{TC}_{\varphi}(x, y)$ to denote the transitive closure of $\varphi(x, y)$. This formula can be written in many different formalisms. For example, if $\varphi(x, y)$ is a conjunction of relational atoms, then $\text{TC}_{\varphi}(x, y)$ can be written as follows in Datalog:

$$\varphi(x, y) \rightarrow \text{TC}_{\varphi}(x, y), \qquad \varphi(x, z), \text{TC}_{\varphi}(z, y) \rightarrow \text{TC}_{\varphi}(x, y).$$

In our system, $\text{TC}_{\varphi}(x, y)$ is written as an SQL query with recursion. Then to deal with an ontology $\mathcal{O}$ containing transitive predicates, the set of inference rules in Table 1 is extended with the following inference rule:

$$(\texttt{A}, \texttt{type}, \texttt{transProp}) : \frac{\{\beta_i(s, o, \bar{x}_i) \wedge p = \texttt{A} \rightarrow \texttt{triple}(s, p, o)\}_{i=1}^{k}}{\text{TC}_{[\bigvee_{i=1}^{k} \exists \bar{x}_i \beta_i]}(s, o) \wedge p = \texttt{A} \rightarrow \texttt{triple}(s, p, o)}.$$

This rule tell us that given a transitive predicate A, we can take any number k of RDB2RDF rules $\beta_i(s, o, \bar{x}_i) \wedge p = \texttt{A} \rightarrow \texttt{triple}(s, p, o)$ for this predicate, and

we can generate a new RDB2RDF rule for A by putting together the conditions $\beta_i(s, o, \bar{x}_i)$ in a formula $\gamma(s, o) = \bigvee_i \exists \bar{x}_i \beta_i(s, o, \bar{x}_i)$, and then using the transitive closure $\mathrm{Tc}_\gamma(s, o)$ of γ in an RDB2RDF rule $\mathrm{Tc}_\gamma(s, o) \wedge p = \mathtt{A} \rightarrow \mathtt{triple}(s, p, o)$. In order for this approach to work, notice that we need to extend the syntax of RDB2RDF rules (1) and (2), so that formulae α and β in them can be arbitrary formulae in a more expressive formalism such as (recursive) Datalog.

Implementing RDB2RDF Mappings as Views. Inspired by our previous work on Ultrawrap [67], every RDB2RDF rule is implemented as a triple-query, that is, as a SQL query which outputs triples. For example, the RDB2RDF rules:

$$\mathtt{order}(s, x_1, x_2, x_3, x_4, \mathtt{1}) \wedge p = \mathtt{type} \wedge o = \mathtt{SuccessfulOrder} \rightarrow \mathtt{triple}(s, p, o)$$
$$\mathtt{order}(s, x_1, x_2, x_3, x_4, \mathtt{2}) \wedge p = \mathtt{type} \wedge o = \mathtt{SuccessfulOrder} \rightarrow \mathtt{triple}(s, p, o)$$

give rise to the following triple-queries:

```
SELECT orderid as S, "type" as P, "SuccessfulOrder" as O FROM order WHERE status = "1"
SELECT orderid as S, "type" as P, "SuccessfulOrder" as O FROM order WHERE status = "2"
```

In practice, the triple-queries may include additional projections in order to support indexes, URI templates, datatypes and languages. However, for readability, we will consider here this simple version of these queries. Then to implement an RDB2RDF mapping, all the class (resp. predicate) RDB2RDF-rules for the same class (resp. predicate) are grouped together to generate a triple-view, that is, a SQL view comprised of the union of the triple-queries for this class (resp. predicate). For instance, in our previous example the following is the triple-view for the class `SuccessfulOrder`:

```
CREATE VIEW SuccessfulOrderView AS
SELECT orderid as S, "type" as P, "SuccessfulOrder" as O FROM order WHERE status = "1"
UNION ALL
SELECT orderid as S, "type" as P, "SuccessfulOrder" as O FROM order WHERE status = "2"
```

SPARQL to SQL Rewriting with RDB2RDF Mappings. The runtime phase executes SPARQL queries on the RDBMS. We reuse Ultrawrap's approach of translating SPARQL queries to SQL queries in terms of the views defined for every class and property, which are denoted as triple-views. Thus, we make maximal use of existing query optimization tools in commercial RDBMS, such as Oracle, to do the SPARQL query execution and rewriting.

Continuing with the example in Sect. 4.2, consider now a SPARQL query which asks for all the Successful Orders: `SELECT ?x WHERE {?x type SuccessfulOrder}`. It is clear that this query needs to be rewritten to ask for the orders with status 1 and 2. The `SuccessfulOrderView` triple-view in Sect. 4.2 implements the mappings to the SuccessfulOrder class which consists of two triple-queries, one each for status = 1 and status = 2. Therefore, it is

sufficient to generate a SQL query in terms of the `SuccessfulOrderView`. Given that a triple-view models a table with three columns, a SPARQL query is syntactically translated to a SQL query in terms of the triple-view. The resulting SQL query is `SELECT t1.s AS x FROM SuccessfulOrderView t1`.

A natural question at this point is whether every SPARQL query has an equivalent SQL query in our context, where RDB2RDF mappings play a fundamental role. In what follows we give a positive answer to this question.

Theorem 8. *Given an RDB2RDF mapping $\mathcal{M}$, every SPARQL query is SQL-rewritable under $\mathcal{M}$.*

The proof that the previous condition holds is by induction on the structure of a SPARQL query P and, thus, it gives us a (naïve) bottom-up algorithm for translating P into an equivalent SQL query Q (given the mapping $\mathcal{M}$). More precisely, in the base case we are given a triple pattern $t = \{$`s p o`$\}$, where each one of its component is either a URI or a literal or a variable. This triple pattern is first translated into a SPARQL query P_t, where each position in t storing a URI or a literal is replaced by a fresh variable, a filter condition is added to ensure that these fresh variables are assigned the corresponding URIs or literals, and a SELECT clause is added to ensure that the output variables of t and P_t are the same. For example, if $t = \{$`?x type SuccessfulOrder`$\}$, then P_t is the following SPARQL query: `SELECT ?x WHERE {?x ?y ?z} FILTER (?y = type && ?z = SuccessfulOrder)`. Then a SQL-rewriting of P_t under $\mathcal{M}$ is computed just by replacing a triple pattern of the form $\{$`?s ?p ?o`$\}$ by a union of all the triple-queries representing the RDB2RDF rules in $\mathcal{M}$, and also replacing the SPARQL filter condition in P_t by a filter condition in SQL.

In the inductive step, we assume that the theorem holds for two SPARQL queries P_1 and P_2.

The proof then continues by presenting rewritings for the SPARQL queries constructed by combining P_1 and P_2 through the operators SELECT, AND (or '.' operator), OPTIONAL, FILTER and UNION, which is done by using existing approaches to translate SPARQL to SQL [5,19].

Cost Model for View Materialization. A common approach for query optimization is to use materialized views [36]. Given that we are implementing RDB2RDF mappings as views, it is a natural to pursue this option. There are three implementation alternatives: (1) Materialize all the views: This approach gives the best query response time. However, it consumes the most space. (2) Materialize nothing: In this approach, every query needs to go to the raw data. However, no extra space is needed. (3) Materialize a subset of the views: Try to find a trade-off between the best query response time and the amount of space required. Note that in the previous Ultrawrap work, only unmaterialized views were considered.

In this section, we present a cost model for these three alternatives. First we must introduce some terminology. We consider ontologies consisting of hierarchy of classes which form a tree with a unique root, where a root class of an ontology

is a class that has no superclasses. Then a leaf class of an ontology is a class that has no subclasses, and the depth of a class is the number of subclass relationships from the class to the root class (notice that there is a unique path from a class to the root class). Moreover, the depth of an ontology is the maximum depth of all classes present in the ontology.

First, we consider the cost of answering a query Q is equal to the number of rows present in the relation used to construct Q. For example, if a relation R has 100 rows, then the cost of the query `SELECT * FROM R` is 100. Second, assume we have a single relation R and that mappings are from a query on the relation R with a selection on an attribute A, to a class in the ontology. For example, consider the relation R is `order`, the attribute A is `status` and the mapping is to the class `SuccessfulOrder`. Finally, we consider a query workload of queries asking for the instances of a class in the ontology, i.e. `SELECT ?x WHERE {?x type C}`, which can be translated into the triple-view implementing the mapping to the class `C`.

Our cost model is the following: If all the views implementing mappings are materialized, the query cost is $n \times N_R \times S(A, R)$ where n is the number of leaf classes underneath the class that is being queried for, N_R is the number of tuples of the relation R in the mapping, and $S(A, R)$ is the selectivity of the attribute A of the relation R in the mapping. The space cost is $N_R + (N_R \times d)$ where d is the depth of the ontology. The reason for this cost is because the number of rows in a materialized view depends on the selectivity of the attribute and the number of leaf classes. Additionally, the sum of all the rows of each triple-view representing the mapping to classes in a particular depth d of an ontology, is equivalent at most to the number of rows of the relation. If no views are materialized, then the query cost is $n \times N_R$, assuming there are no indices. The space cost is simply N_R. The reason for this cost is because to answer a query, the entire relation needs to be accessed n times because there are no indices[12].

The question now is: How can we achieve the query cost of materializing all the views while keeping space to a minimum? Our hypothesis is the following: If a RDBMS rewrites queries in terms of materialized views, then by only materializing the views representing mappings to the leaf classes, the query cost would be $n \times N_R \times S(A, R)$, the same as if we materialized all the views, and the space cost would only be $2 \times N_R$. The rationale is the following: A triple-view representing a mapping to a class, can be rewritten into the union of triple-views representing the mapping to the child classes. Subsequently, a triple-view representing the mapping to any class in the ontology can be rewritten into a union of triple-views representing the mappings to leaf classes of an ontology. Finally, given a set of triple-views representing mappings from a relation to each leaf class of an ontology, the sum of all the rows in the set of triple-views is equivalent to the number of rows in the relation.

[12] In the evaluation, we also consider the case when indices are present.

Given the extensive research of answering queries using views [37] and the fact that Oracle implements query rewriting on materialized views[13], we strongly suspect that our hypothesis will hold. The evaluation provides empirical results supporting our hypothesis.

Evaluation. The evaluation requires benchmarks consisting of a relational database schema and data, ontologies, mappings from the database to ontologies and a query workload. Thus, we created a synthetic benchmark, the *Texas Benchmark*, inspired by the Wisconsin Benchmark [27] and extended the Berlin SPARQL Benchmark (BSBM) Explore Use Case [11]. More details about the benchmarks can be found at http://obda-benchmark.org.

The objective of our experiments is to observe the behavior of a commercial relational database, namely Oracle, and its capabilities of supporting subclass and transitivity reasoning under our proposed approach. The evaluation considered six scenarios: (**all-mat**) all the views are materialized; (**union-leaves**) only views representing mappings to the leaf classes are materialized, implemented with UNION; (**or-leaves**) same as in the previous scenario but with the views implemented with OR instead of UNION, (**union-index**) none of the views, implemented with UNION, are materialized, instead an index on the respective attributes have been added, (**or-index**) same as in the previous scenario but with the views implemented with OR; and (**ontop**) we compare against Ontop, a state of the art OBDA system [61].

An initial assessment suggests the following four expected observations: (1) The fastest execution time is *all-mat*; (2) our hypothesis should hold, meaning that the execution time of *union-leaves* should be comparable, if not equal, to the execution time of *all-mat*; (3) given that the Ontop system generates SQL queries with OR instead of UNION [61], the execution time of *ontop* and *or-index* should be comparable if not equal; (4) with transitivity, the fastest execution time is when the views are materialized.

The experimental results suggest the following. The expected observations (1), (2), (3) and (4) hold. The fastest execution time corresponds to *all-mat*. The execution time of *union-leaves* is comparable, if not equal, to the execution time of *all-mat*, because Oracle was able to rewrite queries in terms of the materialized views. The number of rows examined is equivalent to the number of rows in the views where everything was materialized. This result provides evidence supporting our hypothesis and validates our cost model. Finally the execution time of *ontop* and *or-index* are comparable. It is clear that materializing the view outperforms the non-materialized view for the following reasons: when the view is materialized, the size of the view is known beforehand and the optimizer is able to do a range scan with the index. However, when the view is not materialized, the size is not known therefore the optimizer does a full scan of the table.

[13] http://docs.oracle.com/cd/B28359_01/server.111/b28313/qrbasic.htm.

5 Relational Databases and Semantic Web in Practice

A successfully repeated use case for using Semantic Web technologies with Relational Databases is for data integration. In this approach, an ontology serves as a uniform conceptual federating model, which is accessible to both IT developers and business users. We highlight two challenges: ontology and mapping engineering. We postulate the need of a pay-as-you-go methodology that address these challenges and enables agility.

5.1 A Real World Example

Consider the following real-world example. Executives of a large e-commerce company need to know, "*How many orders were placed in a given month and the corresponding net sales*". Depending on whom they ask they get three different answers. The IT department managing the web site records an order when a customer has checked out. The fulfillment department records an order when it has shipped. Yet the accounting department records an order when the funds charged against the credit card are actually transferred to the company's bank account, regardless of the shipping status. Unaware of the source of the problem the executives are vexed by inconsistencies across established business intelligence (BI) reports.

This is precisely where the use of ontologies to intermediate IT development and business users is valuable. Ontologies serve as a uniform conceptual federated model describing the domain of interest. The long standing relationship between Semantic Web technologies and Relational Databases, specifically the Ontology Based Database Access (OBDA) paradigm and its extension as Ontology Based Data Integration is maturing, and yielding successful applications.

Even though OBDA has been widely researched theoretically, there is still need to understand how to effectively implement OBDA systems in practice.

5.2 Where Do Ontologies and Mappings Come From?

The common definition of OBDA states that given a source relational database, a target ontology and a mapping from the relational database to the ontology, the goal is to answer queries over the target ontology using these three components. From a practical point of view, this begs the question: where does the target ontology and the mappings come from?

Ontology Challenges. Ontology engineering is a challenge by itself. In order to create the target ontology, users can follow traditional ontology engineering methodologies [23,74], using competency questions [8,60], test driven development [43], ontology design patterns [40], etc. Additionally, per standard practices, it is recommended to reuse and extend existing ontologies in domains of interest such as Good Relations for e-commerce[14], FIBO for finance[15], Gist

[14] http://www.heppnetz.de/projects/goodrelations/.
[15] http://www.edmcouncil.org/financialbusiness.

for general business concepts[16], Schema.org[17], etc. In OBDA, the challenge increases because the source database schemas can be considered as additional inputs to the ontology engineering process. Common enterprise application's database schema commonly consist of thousands of tables and tens of thousands of attributes. A common approach is to bootstrap ontologies derived from the source database schemas, known as putative ontologies [65,69]. The putative ontologies can gradually be transformed into target ontologies, using existing ontology engineering methodologies.

Mapping Challenges. Once the target ontology has been created, the source databases can be mapped. The W3C Direct Mapping standard can be used to bootstrap mappings [7]. The declarative nature of W3C R2RML mapping language [25] enables users to state which elements from the source database are connected to the target ontology, instead of writing procedural code. Given that source database schemas are very large, the OBDA mapping challenge is suggestive of an ontology matching problem: the putative ontology of the source database and the target ontology. In addition to 1–1 correspondences between classes and properties, mappings can be complex involving calculations and rules that are part of business logic. For example, the notion of net sales of an order is defined as gross sales minus taxes, discounts given, etc. The discount can be different depending on the type of user. Therefore, a business user needs to provide these definitions before hand. That is why it is hard to automate this process. Another challenge is to create tools that can create and manage mappings [68].

Addressing these challenges is crucial for the success of a data integration project using the OBDA paradigm. However, recall that data integration is a means to an end. The engineering of a target ontology and mappings are the means. Answering business questions are the ends. We observe that target ontologies and mappings are developed in a holistic approach. Given how OWL ontologies are flexible and R2RML mappings are declarative, these elements could enable the incremental development of a target ontology and database mappings. Thus, we argue for a pay-as-you-go methodology for OBDA.

A Pay-as-you-go Methodology for OBDA. We present a methodology to create the target ontology and mappings for an OBDA system, driven by a prioritized list of business questions. The data answering the business questions serve as content of the Business Intelligence (BI) reports that business users require. The objective is to create a target ontology and mappings, that enable to answer a list of business questions, in an incremental manner. After a minimal set of business questions have been successfully modeled, mapped, answered and made into dashboards, then the set of business questions can be extended. The new questions, in turn, may extend the target ontology and new mappings

[16] https://semanticarts.com/gist/.
[17] http://schema.org/.

incrementally added. With this methodology, the target ontology and mappings are developed in an iterative pay-as-you-go approach. Thus, providing an agile methodology to integrate data using the OBDA paradigm because the focus is to provide early and continuous delivery of answers to the business users.

We identify three actors involved throughout the process:

- Business User: a subject matter expert who has knowledge of the business and can identify the list of prioritized business questions.
- IT Developer: a person who has knowledge of databases and knows how the data is interconnected.
- Knowledge Engineer: a person who serves as a communication bridge between Business Users and IT Developers and has expertise in modeling data using ontologies.

Our methodology is divided in two phases: knowledge capture and implementation. Figure 4 summarizes the methodology.

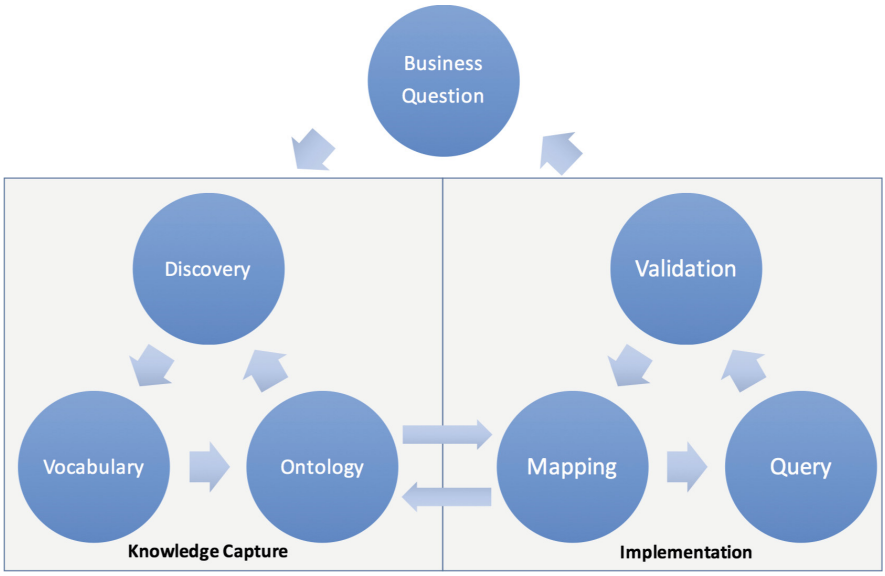


Fig. 4. The pay-as-you-go methodology for OBDA

Knowledge Capture: Discover-Vocabulary-Ontology. The goal of the knowledge capture phase is twofold. The first goal is to extract key concepts and relationships from the set of prioritized business questions. The knowledge engineer works with business users to understand the meaning of extracted concepts and relationships in order to eliminate ambiguity. The second goal is to identify which source database(s) contains data relating to the extracted concepts and relationships. The knowledge engineer takes what has been extracted

with the business users and works with IT developers to identify which tables and attributes are required. This knowledge capture phase is divided in three steps:

- Discovery: Discover the concepts and relationships from the input set of prioritized business questions and identify how the concepts and relationships are connected to the database(s).
- Vocabulary: Identify the business terminology such as preferred labels, alternative labels and natural language definitions for the concepts and relationships.
- Ontology: Formalize the ontology in OWL such that it covers the business questions.

Continuing with our initial example; the knowledge engineer works with business users to understand the meaning of the word "Order". Furthermore, working with IT developers, the knowledge engineer may learn that the Order Management System is the authoritative source for all orders. Within that database, the data relating to orders may be vertically partitioned across a several tables totaling hundreds of attributes. Finally, the attributes required for the calculation of the net sales of an order prove to be only a handful of the hundreds of attributes. The next step is to implement the mappings.

Implementation: Mapping-Query-Validation. The goal of the implementation phase is to enable answering the business questions by connecting the ontology with the data. That is the knowledge engineer takes what was learned from the previous steps and implements the mapping in R2RML. The business questions are implemented as SPARQL queries using the business terminology defined in the target ontology. The R2RML mapping is the input to an OBDA system which will enable the execution of the SPARQL queries. A final step is to validate the results of the queries with business users. To summarize, the implementation phase is divided in three steps:

- Mapping: Implement the mapping in R2RML, given the output of the Discover and Ontology steps. The mapping is then used to setup the OBDA system.
- Query: Business questions are implemented as SPARQL queries using the terminology of the target ontology. The answers to the business questions are the SPARQL results.
- Validation: Confirm that the SPARQL queries return the correct answers.

Continuing with our running example; the result from the knowledge capture phase revealed that the business users considered an order, "an order", if it had shipped or the accounts receivable had been received. The knowledge engineer (the R2RML writer) in conversation with the IT developer identified that requirement as all tuples in the MASTERORDER table where order status is equal to 2 or 3. Therefore, an R2RML mapping consists of the following SQL query:

```
SELECT * FROM MASTERORDER WHERE orderstatus IN (2,3)
```

The definition of net sales of an order is a math formula that uses attributes from the order and ordertax table. This can be represented in the following SQL query:

```
SELECT o.orderid, o.ordertotal - ot.finaltax -
  CASE WHEN o.currencyid in ('USD', 'CAD') THEN o.shippingcost
       ELSE o.shippingcost - ot.shippingtax END AS netsales
FROM order o, ordertax ot
WHERE o.orderid = ordertax.orderid
```

At this point, we can go back to the knowledge capture step for two reasons. If the validation was successful, then we can start another iteration of the approach by soliciting a new set of business questions. On the other hand, if the validation was unsuccessful because the queries did not return the expected results, we can revisit the mappings for that specific fragment. Fixing the problem is now in a compartmentalized section of the ontology and corresponding mappings. Progress is made in an incremental and isolated effort. In worst case the original business logic needs to be revisited and we can go back to the discovery step.

Using this pay-as-you-go methodology for applications across multiple industries is yielding agile results. Development cycles of 1–2 weeks yield new dashboards. All stakeholders are concentrated on a specific task, an agreed upon set of business questions. As development issues arise conversations between the knowledge engineers, business users and IT developers are focused on specific, manageably scoped concepts. The knowledge capture and implementation steps can be accomplished independently. Furthermore, by starting small, the target ontology and mappings are created monotonically. This means that new concepts, relationships and mappings are added without disturbing the work that already has been done. In the case when change to past work is required, it is accomplished without much disruption. The declarative aspect of R2RML mappings, enables focus on what needs to be connected between the source and target instead of writing procedural code and scripts which can be complex to maintain.? Finally, success of each iteration is well defined: answer the business questions.

6 Conclusion

The answer to the question: *How and to what extent can Relational Databases be integrated with the Semantic Web?* comes in three parts:

- **Relational Databases can be directly mapped to RDF and OWL**: Relational Databases can be automatically mapped to the Semantic Web. An OWL ontology can be generated from the relational schema and the relational data can be represented as an RDF graph. This mapping does not loose information, preserves queries, is monotone and is positive semantics preserving. Additionally, it is not possible to have a monotone and full semantics preserving direct mapping.

- **Relational Databases can evaluate and optimize SPARQL queries**: Relational Databases are able to efficiently evaluate SPARQL queries. By implementing the direct mapping using SQL views, relational optimizer exploit two important semantic query optimizations: detection of unsatisfiable conditions and self join elimination.
- **Relational Databases can act as reasoners**: Given a Relational Database, an OWL ontology with inheritance and transitivity, and a mapping between the two, Relational Databases are able to act reasoner. This is possible by implementing the mappings as SQL views and including SQL recursion, materializing a subset of the views based on a cost model, and exploiting existing optimizations such as query rewriting using materialized views.

The results of our research is embodied in a system called Ultrawrap.

6.1 Open Problems

The relationship between Relational Databases and the Semantic Web is via **mappings**. Semantic Web technology provides the following features. OWL Ontologies enable reasoning (**reasoning**). SPARQL queries with variables in the predicate position reveal metadata. This is useful because it enables exploration of the data in case the schema is not known beforehand. Additionally, queries of this form are intrinsic to faceted search (**variable predicate**). Given the graph model of RDF, the latest version of SPARQL, SPARQL 1.1, increased the expressivity and now provides constructs to navigate the graph (**graph traversal**). Another virtue of dealing with graphs is that insertion of data is reduced to adding an edge with a node to the graph. There are no physical requirements to conform to a schema (**dynamic schema**). Finally, data can be easily integrated by simply adding edges between nodes of different graphs (**data integration**).

A goal of our research has been to understand *up to what extent* can Relational Databases be integrated with the Semantic Web. The extent of our research has focused on mappings and reasoning. A remaining question is: can that extent be expanded? And up to where? We call this the Tipping Point problem.

Assume the starting point are legacy relational databases and we want to take advantage of these five features of the Semantic Web (reasoning, variable predicate, graph traversal, dynamic schema, data integration). How much can be subsumed by Relational Database technology before the balance is tipped over and we end up using native Semantic Web technology? What is the tipping point (or points)?

- **Mappings**: The engineering of mappings is still open grounds for research. What mappings patterns can be defined and reused in order to solve a commonly occurring problem [63]? Given that R2RML mappings are represented in RDF, these can be stored in a triplestore, queried and reasoned upon. This opens up potential such as mapping analysis, automatically generating mappings, reusing existing mappings during the engineering of new

mappings, consistency checking of mappings in conjunction with the ontology, adding provenance information to the mappings to support data lineage [22,26,42,45,71]. Additionally, there is a need for tools to support users to create mappings [12,68].

- **Reasoning**: Our research proposed to represent ontological entailments as mappings and implement them as views. Subsequently, a subset of these views are materialized. Open questions remain. What is the state of the art of other RDBMS's optimizers in order to support this approach? How does this approach respond to complex query workloads? The model assumed a read-only database, therefore, what is the cost of maintaining views when the underlying data is updated? Evidence is provided that Relational Databases can act as reasoners for RDFS and Transitivity. Can the expressivity be increased while maintaining efficient computation by the RDBMS optimizer? What is the trade-off between reasoning over relational databases with mappings and using native RDF databases which supports reasoning?
- **Variable Predicate**: For queries with variables in the predicate position, the mapping stipulates that the variable may be bound to the name of any column in the database. These queries are a syntactic construct of higher order logic. Ultrawrap translates these queries into a SQL query consisting of a union for each attribute in the database. This query ends up reading the entire database and suffers a performance penalty. What optimizations can be implemented in order to overcome this issue? What hints can be provided in a query?
- **Graph Traversal**: Regular Path Queries and SPARQL 1.1 property path queries enable pattern-based reachability queries. These types of queries enable the traversal and navigation of the graph. A natural question is how much of SQL recursion can be used to implement these types of queries?
- **Dynamic Schema**: Relational Databases have a fixed schema. Insertion of data needs to adhere to the schema. A schema needs to be altered in case new data is inserted which does not adhere to the schema. Can a Relational Database become hybrid graph/relational database? What effect does the sparsity of data have? What is the best storage manager (column vs row store)?
- **Data Integration**: When it comes to integrate disparate databases, one approach is to extract the relational data, transform it physically to RDF and then load it into a RDF database (ETL). Another approach is to federate queries. In other words, legacy data continues to reside in the relational databases and queries are sent to each source (Federation). Which approach is practical? Depending on what? Can hybrid system be efficient?

An overarching theme is the need to create systematic and real-world benchmarks in order to evaluate different solutions for these features.

These open questions provide a roadmap to further expand the extent that Relational Databases can be integrated with the Semantic Web.

References

1. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 411–422 (2007)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
3. Ahmed, R., Lee, A., Das, D.: Join predicate push-down optimizations. US Patent 7,945,562, May 17 2011
4. Allemang, D., Hendler, J.A.: Semantic Web for the Working Ontologist - Effective Modeling in RDFS and OWL, 2nd edn. Morgan Kaufmann, San Francisco (2011)
5. Angles, R., Gutierrez, C.: The expressive power of SPARQL. In: Sheth, A., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 114–129. Springer, Heidelberg (2008). doi:10.1007/978-3-540-88564-1_8
6. Arenas, M., Barceló, P., Libkin, L., Murlak, F.: Foundations of Data Exchange. Cambridge University Press, Cambridge (2014)
7. Arenas, M., Bertails, A., Prud'hommeaux, E., Sequeda, J.: Direct mapping of relational data to RDF. W3C Recomendation, 27 September 2012. http://www.w3.org/TR/rdb-direct-mapping/
8. Azzaoui, K.: Scientific competency questions as the basis for semantically enriched open pharmacological space development. Drug Discov. Today **18**, 843–852 (2013)
9. Baader, F., Brandt, S., Lutz, C.: Pushing the el envelope. In: IJCAI (2005)
10. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, Cambridge (2003)
11. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. Int. J. Semant. Web Inf. Syst. **5**(2), 1–24 (2009)
12. Blinkiewicz, M., Bąk, J.: SQuaRE: a visual approach for ontology-based data access. In: Li, Y.-F., Hu, W., Dong, J.S., Antoniou, G., Wang, Z., Sun, J., Liu, Y. (eds.) JIST 2016. LNCS, vol. 10055, pp. 47–55. Springer, Cham (2016). doi:10.1007/978-3-319-50112-3_4
13. Brickley, D., Guha, R.: RDF vocabulary description language 1.0: RDF schema, W3C recommendation, February 2004
14. Broekstra, J., Kampman, A., Harmelen, F.: Sesame: a generic architecture for storing and querying RDF and RDF schema. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 54–68. Springer, Heidelberg (2002). doi:10.1007/3-540-48005-6_7
15. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Data complexity of query answering in description logics. Artif. Intell. **195**, 335–360 (2013)
16. Calvanese, D., Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: the DL-Lite family. J. Autom. Reason. **39**(3), 385–429 (2007)
17. Calvanese, D., Giacomo, G.D., Lembo, D., Lenzerini, M., Rosati, R.: EQL-Lite: effective first-order query processing in description logics. In: IJCAI, pp. 274–279 (2007)
18. Chakravarthy, U.S., Grant, J., Minker, J.: Logic-based approach to semantic query optimization. ACM Trans. Database Syst. **15**(2), 162–207 (1990)
19. Chebotko, A., Lu, S., Fotouhi, F.: Semantics preserving SPARQL-to-SQL translation. Data Knowl. Eng. **68**(10), 973–1000 (2009)

20. Cheng, Q., Gryz, J., Koo, F., Leung, T.Y.C., Liu, L., Qian, X., Schiefer, K.B.: Implementation of two semantic query optimization techniques in DB2 universal database. In: VLDB, pp. 687–698 (1999)
21. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An efficient SQL-based RDF querying scheme. In: Proceedings of the 31st International Conference on Very Large Data Bases, pp. 1216–1227 (2005)
22. Civili, C., Mora, J., Rosati, R., Ruzzi, M., Santarelli, V.: Semantic analysis of R2RML mappings for ontology-based data access. In: Ortiz, M., Schlobach, S. (eds.) RR 2016. LNCS, vol. 9898, pp. 25–38. Springer, Cham (2016). doi:10.1007/978-3-319-45276-0_3
23. Corcho, Ó., Fernández-López, M., Gómez-Pérez, A.: Methodologies, tools and languages for building ontologies: where is their meeting point? Data Knowl. Eng. **46**(1), 41–64 (2003)
24. Cudré-Mauroux, P., et al.: NoSQL databases for RDF: an empirical evaluation. In: Alani, H., et al. (eds.) ISWC 2013. LNCS, vol. 8219, pp. 310–325. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41338-4_20
25. Das, S., Sundara, S., Cyganiak, R.: R2RML: RDB to RDF mapping language. W3C Recomendation, 27 September 2012. http://www.w3.org/TR/r2rml/
26. Medeiros, L.F., Priyatna, F., Corcho, O.: MIRROR: automatic R2RML mapping generation from relational databases. In: Cimiano, P., Frasincar, F., Houben, G.-J., Schwabe, D. (eds.) ICWE 2015. LNCS, vol. 9114, pp. 326–343. Springer, Cham (2015). doi:10.1007/978-3-319-19890-3_21
27. DeWitt, D.J.: The Wisconsin benchmark: past, present, and future. In: The Benchmark Handbook, pp. 119–165 (1991)
28. Donini, F., Lenzerini, M., Nardi, D., Nutt, W., Schaerf, A.: An epistemic operator for description logics. Artif. Intell. **100**(1–2), 225–274 (1998)
29. Donini, F.M., Nardi, D., Rosati, R.: Description logics of minimal knowledge and negation as failure. ACM Trans. Comput. Log. **3**(2), 177–225 (2002)
30. Elliott, B., Cheng, E., Thomas-Ogbuji, C., Ozsoyoglu, Z.M.: A complete translation from SPARQL into efficient SQL. In: Proceedings of the 2009 International Database Engineering & Applications Symposium, pp. 31–42 (2009)
31. Franke, C., Morin, S., Chebotko, A., Abraham, J., Brazier, P.: Distributed semantic web data management in HBase and MySQL cluster. In: Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing, pp. 105–112 (2011)
32. Glimm, B., Hogan, A., Krotzsch, M., Polleres, A.: OWL-LD. http://semanticweb.org/OWLLD/
33. Gray, A.J., Gray, N., Ounis, I.: Can RDB2RDF tools feasibily expose large science archives for data integration? In: Aroyo, L., et al. (eds.) ESWC 2009. LNCS, vol. 5554, pp. 491–505. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02121-3_37
34. Grimm, S., Motik, B.: Closed world reasoning in the semantic web through epistemic operators. In: OWLED (2005)
35. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: combining logic programs with description logic. In: WWW, pp. 48–57 (2003)
36. Gupta, A., Mumick, I.S., Views, M.: Techniques, Implementations, and Applications. MIT Press, Cambridge (1999)
37. Halevy, A.Y.: Answering queries using views: a survey. VLDB J. **10**(4), 270–294 (2001)
38. Harris, S., Seaborne, A.: SPARQL 1.1 query language. W3C Recommendation, 21 March 2013. http://www.w3.org/TR/sparql11-query/
39. Hendler, J.: RDFS 3.0. In: W3C Workshop - RDF Next Steps (2010)

40. Hitzler, P., Gangemi, A., Janowicz, K., Krisnadhi, A., Presutti, V. (eds.): Ontology Engineering with Ontology Design Patterns - Foundations and Applications. Studies on the Semantic Web, vol. 25. IOS Press (2016)
41. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL querying of large RDF graphs. PVLDB **4**(11), 1123–1134 (2011)
42. Jiménez-Ruiz, E.: BootOX: practical mapping of RDBs to OWL 2. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 113–132. Springer, Cham (2015). doi:10.1007/978-3-319-25010-6_7
43. Keet, C.M., **L**awrynowicz, A.: Test-driven development of ontologies. In: Sack, H., Blomqvist, E., d'Aquin, M., Ghidini, C., Ponzetto, S.P., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9678, pp. 642–657. Springer, Cham (2016). doi:10.1007/978-3-319-34129-3_39
44. Ladwig, G., Harth, A.: CumulusRDF: linked data management on nested key-value stores. In: 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011) (2011)
45. Lembo, D., Mora, J., Rosati, R., Savo, D.F., Thorstensen, E.: Mapping analysis in ontology-based data access: algorithms and complexity. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9366, pp. 217–234. Springer, Cham (2015). doi:10.1007/978-3-319-25007-6_13
46. Lenzerini, M.: Data integration: a theoretical perspective. In: PODS, pp. 233–246 (2002)
47. Lutz, C., Seylan, İ., Toman, D., Wolter, F.: The combined approach to OBDA: taming role hierarchies using filters. In: Alani, H., et al. (eds.) ISWC 2013. LNCS, vol. 8218, pp. 314–330. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41335-3_20
48. MahmoudiNasab, H., Sakr, S.: An experimental evaluation of relational RDF storage and querying techniques. In: Yoshikawa, M., Meng, X., Yumoto, T., Ma, Q., Sun, L., Watanabe, C. (eds.) DASFAA 2010. LNCS, vol. 6193, pp. 215–226. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14589-6_22
49. Mehdi, A., Rudolph, S., Grimm, S.: Epistemic querying of OWL knowledge bases. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., Leenheer, P., Pan, J. (eds.) ESWC 2011. LNCS, vol. 6643, pp. 397–409. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21034-1_27
50. Motik, B., Grau, B.C., Horrocks, I., Wu, Z., amd Carsten Lutz, A.F.: Owl 2 web ontology language profiles, 2nd edn., W3C recommendation, December 2012
51. Motik, B., Horrocks, I., Sattler, U.: Bridging the gap between OWL and relational databases. J. Web Semant. **7**(2), 74–89 (2009)
52. Muñoz, S., Pérez, J., Gutierrez, C.: Simple and efficient minimal RDFS. J. Web Semant. **7**(3), 220–234 (2009)
53. Neumann, T., Weikum, G.: The RDF-3x engine for scalable management of RDF data. VLDB J. **19**(1), 91–113 (2010)
54. Ortiz, M., Šimkus, M.: Reasoning and query answering in description logics. In: Eiter, T., Krennwallner, T. (eds.) Reasoning Web 2012. LNCS, vol. 7487, pp. 1–53. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33158-9_1
55. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. **34**(3), 16 (2009)
56. Pinto, F.D., Lembo, D., Lenzerini, M., Mancini, R., Poggi, A., Rosati, R., Ruzzi, M., Savo, D.F.: Optimizing query rewriting in ontology-based data access. In: EDBT (2013)
57. Poggi, A., Lembo, D., Calvanese, D., Giacomo, G.D., Lenzerini, M., Rosati, R.: Linking data to ontologies. J. Data Semant. **10**, 133–173 (2008)

58. Priyatna, F., Corcho, Ó, Sequeda, J.: Formalisation and experiences of R2RML-based SPARQL to SQL query translation using Morph. In: 23rd International World Wide Web Conference, WWW 2014, Seoul, 7–11 April 2014, pp. 479–490 (2014)
59. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Recommendation 15 January 2008. http://www.w3.org/TR/rdf-sparql-query/
60. Ren, Y., Parvizi, A., Mellish, C., Pan, J.Z., Deemter, K., Stevens, R.: Towards competency question-driven ontology authoring. In: Presutti, V., d'Amato, C., Gandon, F., d'Aquin, M., Staab, S., Tordai, A. (eds.) ESWC 2014. LNCS, vol. 8465, pp. 752–767. Springer, Cham (2014). doi:10.1007/978-3-319-07443-6_50
61. Rodríguez-Muro, M., Kontchakov, R., Zakharyaschev, M.: Ontology-based data access: *Ontop* of databases. In: Alani, H., et al. (eds.) ISWC 2013. LNCS, vol. 8218, pp. 558–573. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41335-3_35
62. Sequeda, J.: On the semantics of R2RML and its relationship with the direct mapping. In: Proceedings of the ISWC 2013 Posters & Demonstrations Track, Sydney, 23 October 2013, pp. 193–196 (2013)
63. Sequeda, J., Priyatna, F., Villazón-Terrazas, B.: Relational database to RDF mapping patterns. In: Proceedings of the 3rd Workshop on Ontology Patterns, Boston, 12 November 2012
64. Sequeda, J.F.: Integrating relational databases with the semantic web. IOS Press (2016). https://repositories.lib.utexas.edu/bitstream/handle/2152/30537/SEQUEDA-DISSERTATION-2015.pdf
65. Sequeda, J.F., Arenas, M., Miranker, D.P.: On directly mapping relational databases to RDF and OWL. In: WWW, pp. 649–658 (2012)
66. Sequeda, J.F., Arenas, M., Miranker, D.P.: OBDA: query rewriting or materialization? In practice, both!. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 535–551. Springer, Cham (2014). doi:10.1007/978-3-319-11964-9_34
67. Sequeda, J.F., Miranker, D.P.: Ultrawrap: SPARQL execution on relational data. J. Web Semant. **22**, 19–39 (2013)
68. Sequeda, J.F., Miranker, D.P.: Ultrawrap mapper: a semi-automatic relational database to RDF (RDB2RDF) mapping tool. In: Proceedings of the ISWC 2015 Posters & Demonstrations Track co-located with the 14th International Semantic Web Conference (ISWC-2015), Bethlehem, 11 October 2015
69. Sequeda, J.F., Tirmizi, S.H., Corcho, O., Miranker, D.P.: Survey of directly mapping SQL databases to the semantic web. Knowl. Eng. Review **26**(4), 445–486 (2011)
70. Shenoy, S.T., Ozsoyoglu, Z.M.: A system for semantic query optimization. In: SIGMOD, pp. 181–195 (1987)
71. Sicilia, Á., Nemirovski, G.: AutoMap4OBDA: automated generation of R2RML mappings for OBDA. In: Blomqvist, E., Ciancarini, P., Poggi, F., Vitali, F. (eds.) EKAW 2016. LNCS (LNAI), vol. 10024, pp. 577–592. Springer, Cham (2016). doi:10.1007/978-3-319-49004-5_37
72. Tao, J., Sirin, E., Bao, J., McGuinness, D.L.: Integrity constraints in OWL. In: AAAI (2010)
73. Tirmizi, S.H., Sequeda, J., Miranker, D.: Translating SQL applications to the semantic web. In: Bhowmick, S.S., Küng, J., Wagner, R. (eds.) DEXA 2008. LNCS, vol. 5181, pp. 450–464. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85654-2_40
74. Uschold, M., Gruninger, M.: Ontologies: principles, methods and applications. Knowledge Eng. Review **11**(2), 93–136 (1996)

75. Weaver, J., Hendler, J.A.: Parallel materialization of the finite RDFS closure for hundreds of millions of triples. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 682–697. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04930-9_43
76. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proc. VLDB Endow. **1**(1), 1008–1019 (2008)
77. Wilkinson, K.: Jena property table implementation. Technical report HPL-2006-140, HP Laboratories (2006)