The VLDB Journal

Connecting Crowd-sourced Spatial Information to the Data Web with Sparqlify

--Manuscript Draft--

Manuscript Number:	
Full Title:	Connecting Crowd-sourced Spatial Information to the Data Web with Sparqlify
Article Type:	SI: Structured, Social and Crowd-sourced Data on the Web
Keywords:	SPARQL; SQL; Views; RDB2RDF; VGI; Data Integration; Geo
Corresponding Author:	Jens Lehmann University of Leipzig GERMANY
Corresponding Author Secondary Information:	
Corresponding Author's Institution:	University of Leipzig
Corresponding Author's Secondary Institution:	
First Author:	Claus Stadler
First Author Secondary Information:	
Order of Authors:	Claus Stadler
	Jörg Unbehauen
	Jens Lehmann
	Sören Auer
Order of Authors Secondary Information:	
Abstract:	Crowdsourcing, i.e. aggregating and integrating small contributions from distributed groups of people, has proven to be a very effective way to generate comprehensive knowledge bases. An extremely valuable crowdsourced resource is OpenStreetMap, where thousands of contributors create an online map of the world. However, OpenStreetMap has limitations with regard to the structuring of data, its coherence and the versatility of its use beyond maps creation. We argue, that a synergistic combination of relational data management and Linked Data can help to alleviate these limitations. With the LinkedGeoData project we made a first step in this direction. In this article, we present Sparqlify - the novel approach used in LinkedGeoData for mapping relational data to RDF and Linked Data. Sparqlify is based on an intuitive mapping language, which is inspired by SPARQL construct queries and SQL view definitions. It is aimed at high scalability reaching to billions of data items. This is achieved by directly translating one SPARQL query into exactly one efficiently executable SQL query on the underlying relational database schema, thus pushing the query execution workload efficiently into the RDBMS. Sparqlify is schema and application agnostic and can be easily used in other RDB2RDF scenarios. Our evaluation shows that Sparqlify is scalable and able to deal with query workloads for which other systems currently fail.

Noname manuscript No.
(will be inserted by the editor)

Connecting Crowd-sourced Spatial Information to the Data Web with Sparqlify

Bridging the gap between crowd-sourced data management and data publishing

Claus Stadler · Jörg Unbehauen · Jens Lehmann · Sören Auer

Received: date / Accepted: date

Abstract Crowdsourcing, i.e. aggregating and integrating small contributions from distributed groups of people, has proven to be a very effective way to generate comprehensive knowledge bases. An extremely valuable crowdsourced resource is *OpenStreetMap*, where thousands of contributors create an online map of the world. However, OpenStreetMap has limitations with regard to the structuring of data, its coherence and the versatility of its use beyond maps creation. We argue, that a synergistic combination of relational data management and Linked Data can help to alleviate these limitations. With the *LinkedGeoData* project we made a first step in this direction. In this article, we present *Sparqlify* – the novel approach used in LinkedGeoData for mapping relational data to RDF and Linked Data. Sparqlify is based on an intuitive mapping language, which is inspired by SPARQL construct queries and SQL view definitions. It is aimed at high scalability reaching to billions of data items. This is achieved by directly translating one SPARQL query into *exactly one* efficiently executable SQL query on the underlying relational database schema, thus pushing the query execution workload efficiently into the RDBMS. Sparqlify is schema and application agnostic and can be easily used in other RDB2RDF scenarios. Our evaluation shows that Sparqlify is scalable and able to deal with query workloads for which other systems currently fail.

This work was supported by grants from the European Union's 7th Framework Programme provided for the projects GeoKnow (GA no. 318159) and LOD2 (GA no. 257943).

AKSW Research Group, University of Leipzig
Augustusplatz 10, 04109 Leipzig, Germany
E-mail: {cstadler,unbehauen,lehmann,auer}@informatik.uni-leipzig.de

1 Introduction

Crowdsourcing, i.e. aggregating and integrating small contributions from distributed groups of people, has proven to be a very effective way to generate comprehensive knowledge bases. Probably the most prominent example is *Wikipedia*, where crowd contributions are pieces of encyclopedic text ranging from whole articles (or parts of them) to the correction of small typos. Another, extremely valuable and popular crowdsourced resource is *OpenStreetMap*[1], where thousands of contributors create an online map of the world. OpenStreetMap contributors are drawing (or altering) various types of lines and points on a visible map and may annotate them using attribute-value pairs. However, these additions are translated into additions to and deletions of data to a relatively simple relational database schema. Many millions of such small contributions result in a database of enormous volume[2]. The map is finally created by retrieving all points and lines in a certain area and rendering raster image tiles for that area, based on style definitions stored for certain attribute-value combinations. The elements of OpenStreetMap, i.e. database schema and map rendering, were designed to provide maximum flexibility. For example, arbitrary attribute-value combinations can be created, thus accommodating future and unforeseen usage scenarios. If, for example, someone decides to add glass recycling boxes to the map, this can be easily accommodated by adding the new attribute-value annotation 'recycling=glas' to the location point (identified by its longitude/latitude coordinates). As a result, OpenStreetMap is a prototypical example of a *crowdsourced big data application* comprising large *volume* (300 GB), high *velocity* (hundreds of changes per second), and variety (more than 10.000 attributes and many millions of different attribute-value combinations). However, crowdsourced applications such as OpenStreetMap reveal a number of limitations:

- *Structure.* Despite OpenStreetMap using a relational database schema, large parts of the structure of the data are implicitly represented using attribute-value annotations. These annotations and their structure are not well exposed and accessible by users and applications.
- *Coherence.* There is an enormous potential to connect the data in OpenStreetMap with other data sources, such as *Geonames*, *Wikipedia/DBpedia* or the wealth of open governmental data being recently

[1] `http://openstreetmap.org`
[2] 300 GB plain PostGIS database and 850 GB with typical spatial and other index structures.

published. However, OpenStreetMap does not provide any stable, public, de-referenceable identifiers for its data elements (i.e. points, lines and annotations).
– *Versatility.* OpenStreetMap contains an enormous wealth of data, which can be used for applications beyond map creation. For example, restaurants are tagged with the cuisine they offer, there are opening hours for shops and specific information about historic landmarks. Providing better access to this wealth of information can substantially increase the versatility of OpenStreetMap for various other usage scenarios.

We argue, that a synergistic combination of relational data management and Linked Data can help to alleviate these limitations. With *LinkedGeoData* [2,17] we made a first step in this direction. However, LinkedGeoData was curbed by the sheer amount of data. Due to the limitations of current triple store technology, it is not possible to provide Linked Data access to the whole OpenStreetMap data. We focused on 'interesting' data, however, the decision which parts of the OpenStreetMap data are interesting is difficult and will always result in data important for a particular use case to be excluded. Moreover, it turned out that the time required for the RDF extraction process is a severe bottleneck, in particular due to the high update frequency, schema changes and general growth of OpenStreetMap.

As a solution to this problem, we present in this article our approach for mapping relational data to RDF and Linked Data dubbed *Sparqlify*[3]. Sparqlify is based on an intuitive mapping language, which is inspired by SPARQL construct queries and SQL view definitions. Sparqlify is aimed at high scalability reaching to many billions of data items. This is achieved by directly translating one SPARQL query into *exactly one* efficiently executable SQL query on the underlying relational database schema, thus pushing the query execution workload efficiently into the RDBMS. In order to perform this translation efficiently, Sparqlify employs an elaborated view candidate selection mechanism. The Sparqlify approach is schema and application agnostic and can be easily used in other RDB-RDF mapping scenarios. Due to the expressiveness of its mapping language which is also able to deal with database schemas containing semi-structured elements (e.g. attribute value tables), Sparqlify is well suited for connecting large-scale crowdsourcing applications with the Web of Data. The evaluation of our implementation shows, that Sparqlify is very scalable and able to deal with query workloads where other systems currently fail.

Our contributions are in particular:

– We show that (relational) data management and (RDF) data publication can be efficiently combined. Rather than resorting to one one of these worlds, we demonstrate how a hybrid approach can simplify data management, application development and data publication. This also enables a 'pay-as-you-go' approach, in which you can add more structure, including fine-grained ontologies, to crowdsourced data if required.
– We present a generic formalization that underpins efficient RDB-RDF rewriting approaches. The formalization can serve as foundation for other mapping approaches, including the W3C RDB2RDF working group and their work on formalizing the semantics of R2RML[4].
– We implemented our approach in the *Sparqlify* tool and present with its use in LinkedGeoData a case study that demonstrates the feasibility in very large-scale crowdsourcing applications.
– We demonstrate that Sparqlify is more scalable than current state-of-the-art tools on existing benchmarks.

The article is structured as follows: We introduce preliminaries related to RDF as well as the SPARQL and SQL algebra in Section 2. In Section 3, we give an overview of the LinkedGeoData project. We outline the Sparqlify concept in Section 4, which serves as on overview for the technical part of the article. The Sparqlify mapping language (Sparqlify-ML) is described in Section 5. After that, we present the SPARQL to SQL rewriting approach in Section 6. The process of selecting, which views defined in Sparqlify-ML are relevant for a SPARQL query is defined in Section 7. We then present a relational algebra for RDB2RDF mappings in Section 8, which can also serve as formal foundation for other approaches. Our implementation of the formal framework is described in Section 9 along with performance optimizations for view selection via constraints. The impact of Sparqlify in the LinkedGeoData use case is outlined in Section 10. We evaluate the performance and scalability of Sparqlify in Section 11 using the *BSBM* and *SP2* benchmarks as well as LinkedGeoData itself. We discuss related work in Section 12 and conclude with an outlook on future work in Section 13.

2 Preliminaries

In this section, we introduce some basic notions for RDF, as well as brief comments on the SPARQL and

[3] `https://github.com/AKSW/Sparqlify`

[4] `http://www.w3.org/TR/r2rml/`

SQL algebras. We refer to [5,7,13] for comprehensive introductions of the respective topics.

Throughout the article, we generally use upper case letters to denote sets, lower case letters to denote elements and relations, and calligraphic letters to denote the set containing all elements of a specific type. In particular, we use the following abbreviations for important sets (partially based on work from [7]):

- $\mathcal{U}$ is the set of URIs
- $\mathcal{B}$ is the set of all blank nodes
- $\mathcal{L}$ is the set of all literals
- $\mathcal{V}$ is the set of all query variables
- $\mathcal{T}$ is the set of all *RDF terms*, defined as $\mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$.

Most modern RDF stores operate on quads, which comprise a graph component in addition to the components of a regular RDF triple. For this reason, we base our formalisation on quads instead of triples.

Definition 1 (RDF Graph and Term Relation) An *RDF graph* is a subset of $(\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$. An *RDF named graph* is a subset of $(\mathcal{U} \times \mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$. Furthermore, we define an *RDF term relation* (short *RDF relation*) as a subset of $T \times T \times T \times T$.

Conceptually, Sparqlify maps relational data into an RDF term relation, and the user has to ensure that his mapping definitions result in an RDF named graph.

Definition 2 (Quad and Quad Pattern) We refer to an element $q = (g, s, p, o)$ of an RDF relation as a *quad*. The four components of a quad are called its *graph*, *subject*, *predicate* and *object* in that order. We use the notion q_i with $1 \leq i \leq 4$ and $i \in \mathcal{N}$ to refer to a quad's i-th component, i.e. $q_1 = g$, $q_2 = s$, $q_3 = p$, $q_4 = o$. A *quad pattern* has the same structure as a quad, but may additionally have variables, i.e. elements of $\mathcal{V}$, for each one of its components.

Our approach to rewriting of SPARQL queries is based on *algebraic expression tree* (AET) transformations. For both SPARQL and SQL there exist algebraic representations for the structure of queries besides their syntactic representation, see e.g. [5,7] and the illustration in Figure 1. We do not describe those algebras and their differences in detail here as we use a set of operators that is common to both. The following are the most important elements of the SPARQL 1.0 algebra[5]. Note that the term *pattern* refers to any expression being composed from the following components.

- JOIN(l, r): A join between two patterns l and r.
- LEFT-JOIN(l, r): A left join between two patterns l and r.
- UNION(u_1, ..., u_n): The union of the patterns u_1, ..., u_n.
- FILTER(expr, a): A selection on the pattern a, which only retains those rows of the result set, that satisfy the predicate *expr*.
- PROJECT(a, vars): A projection of the pattern a, which only retains the columns mentioned in *vars*.
- DISTINCT(a): The operator that removes duplicate result solutions.

The SQL algebra is very similar in the sense that it uses the same operators, which are applied, however, to relations (instead of patterns) as primitives. In fact, in terms of expressiveness both algebras have been shown to be equivalent [12].

Syntax	*Algebra Expression*

`SELECT DISTINCT ?s ?o {`
`  ?s ?t ?s .`
`  ?s ex:interest ?o .`
`}`
	`Distinct (`
	`  Project ((?s ?o), {`
	`  QuadPatternSet({`
	`  ?s ?t ?s .`
	`  ?s ex:interest ?o`
	`  })`
	`}) )`

Fig. 1: Juxtaposition of a SPARQL query and its AET. The IRI for the default graph has been omitted in the quads.

3 LinkedGeoData Overview

The LinkedGeoData project aims at converting OpenStreetMap data to RDF and connect it to other knowledge bases in the web of data. We first describe OpenStreetMap as crowdsourced community project for collecting spatial data and then give an overview on LinkedGeoData [17] – the RDF conversion of OpenStreetMap.

3.1 OpenStreetMap Data and Community

OpenStreetMap is a collaborative project to create a free editable map of the whole world. It was inspired by Wikipedia and as such it provides well known wiki features such as an edit-tab and a full revision history of the edits. However, rather than editing articles, users edit spatial entities. The three fundamental ones are:

- *Nodes* are the most primitive entities and represent geographic points with a latitude and longitude relative to the WGS84 reference system.
- *Ways* are entities that comprise a list of at least two node references associated with them. Depending on whether the first reference equals the last one, a way is called *closed* or *open*, respectively.

[5] `http://w3.org/TR/rdf-sparql-query/`

– *Relations* relate points, ways and potentially other relations to each other, thereby forming complex objects. Multipolygons are modelled with relations.

Each of these entities has a numeric identifier (called *OSM ID*), a set of generic attributes, and most importantly is described using a set of key-value pairs, known as *tags*.

An example of a relation is the administrative boundary of Germany having the OSM identifier 51477[6]. It comprises more than 1,000 ways, which represent certain segments of the border of Germany; the border with Luxembourg e.g. is composed of approx. 40 way segments. The relation currently has about 30 associated tag-value pairs, which, for example, contain translations of Germany in different languages. One of those tag-value pairs (`boundary=administrative`) indicates that this relation represents an administrative boundary. This information is used by the OSM map renderer to decide how this relation should be displayed on the map. Further tags are used to indicate timezone, currency, and the ISO country code. The relation also has a few metadata entries (such as the timestamp of the last edit and the last editor) attached.

OpenStreetMap data is stored in a PostgreSQL relational database. The data is also published as complete dumps of the database in an XML format on a weekly basis. It currently accounts for more than 23GB of Bzip2 compressed data. In minutely, hourly and daily intervals changesets are published, which can be used to synchronize a local deployment of the data with the OSM database. The dumps as well as the changesets can be processed with the *Osmosis* tool.

OpenStreetMap's community has build a variety different authoring interfaces. These include the online editor *Potlatch*, which is implemented in Flash and accessible directly via the edit tab at the OSM map view, as well as the desktop applications *JOSM*, *Merkaartor* and *Mapzen*. The editors use complementary external services and data such as *Yahoo! satellite imagery* or *Web Map Services* (WMS). Additionally, users can upload GPS traces which serve as raw material for modelling the map. A large number of different rendering services are available for rendering of raster, vector and even 3D maps on different zoom levels. The *Mapnik* renderer is used by OpenStreetMap as default and re-renders tiles in certain intervals.

Since the use of tags (i.e. key-value annotations) is not restricted, but governed by an agile community process, it is important to obtain an overview on emerging tags and tag values possibly specific to a certain region. Services such as *TagWatch*[7] periodically compute tag statistics for different areas. In order for the data to be machine interpretable, as for instance for map rendering, contributors must follow certain editing standards and conventions[8].

OSM is licensed under the Open Database License[9]. The growth of OpenStreetMap data has been enormous (cf. Table 1): Since the founding in July 2004 until now, 1.5 billion nodes, about 150 million ways and 1.5 million relations have been contributed by the users. Some of the data was imported form public domain datasources such as TIGER for US, AND Automotive Navigation Data for The Netherlands, and GeoBase data from the Canadian government.

3.2 Previous RDF Conversion Approach

The goal of the LinkedGeoData project is to provide rich, open, and integrated spatial data to the Semantic Web using OpenStreetMap as its base. This is analogous to the well known DBpedia project, which follows a similar approach based on Wikipedia. The necessary work for reaching this goal comprises the conversion of OSM data to RDF, the interlinking with other knowledge bases, the dissemination of the resulting data, and keeping the datasets up-to-date. Here, we only focus on the RDF coversion approach. We give a summary of the previous LinkedGeoData architecture as described in [17] and point out the weaknesses of that approach and requirements for extracting structures from crowdsourced information in OpenStreetMap.

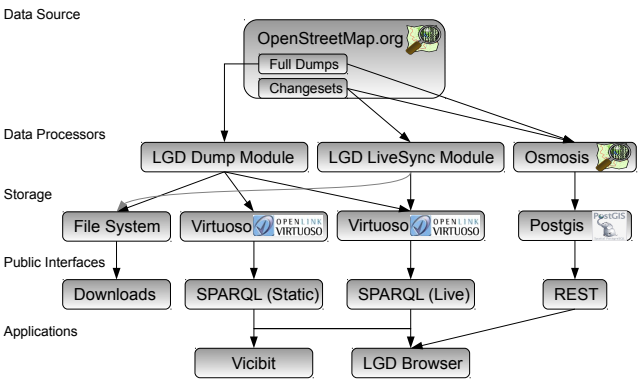

Fig. 2: Overview of LinkedGeoData's architecture before the introduction of Sparqlify.

The previous architecture of LinkedGeoData is illustrated in Figure 2. Data from OpenStreetMap is pro-

[6] `http://openstreetmap.org/browse/relation/51477`

[7] `http://tagwatch.stoecker.eu/`

[8] `http://wiki.openstreetmap.org/wiki/Map_Features`

[9] `http://www.opendatacommons.org/licenses/odbl/`

Category	June 2009	April 2010	May 2011	Aug 2012	Growth (Jun 09 - Aug 12)
Users (Thousands)	127	261	397	793	+ 524%
Uploaded GPS points (Millions)	915	1500	2298	3094	+ 238%
Nodes (Millions)	374	600	1073	1578	+ 322%
Ways (Millions)	30	48	92	150	+ 400%

Table 1: OpenStreetMap statistics 2009 - 2012.
(Obtained from `http://www.openstreetmap.org/stats/data_stats.html` at the specified months.)

cessed on different routes: The *LGD Dump Module* converts an OSM planet file to RDF and loads the data into a triple store. This data is then available via a *static SPARQL endpoint*. A copy of that triple store serves as the initial basis for the *live SPARQL endpoint*. The *LGD Live Sync Module* downloads minutely changesets from OpenStreetMap, and computes corresponding changesets on the RDF level in order to update that triple store accordingly. By publishing these RDF changesets, data consumers can sync their triple store with LinkedGeoData. In this architecture, however, we were not able to load all OSM entities into the SPARQL endpoint due to performance reasons. In addition, the RDF conversion process required frequent changes of the mappings, which meant that a high number of RDF triples required updating resulting in large changesets. The new Sparqlify based architecture, which we describe in detail in Section 10 avoids those problems by using the virtual RDF views of Sparqlify without costly conversion processes. Furthermore, the new architecture allows to directly re-use the existing OpenStreetMap tool chain including PostGIS and Osmosis. For data access LinkedGeoData offers downloads, a REST API interface, Linked Data, and SPARQL endpoints. The *REST API* provides limited query capabilities for RDFized data about *all* nodes and ways of OpenStreetMap.

In summary, our requirements for enabling the extraction of structural data from the crowdsourced spatial information in OpenStreetMap are high scalability, high flexibility and low maintenance effort. In the next sections, we will describe Sparqlify in detail and then revisit the LinkedGeoData use case in Section 10 to explain in detail how the above requirements can be addressed via Sparqlify.

4 The Sparqlify Concept

The challenges encountered with LinkedGeoData clearly showed, that ETL style approaches based on the conversion of all of OpenStreetMap's data to RDF have severe deficiencies. For instance, the RDF conversion process is very time consuming for large-scale, crowdsourced data. Furthermore, changes in data modelling require many changes in the extracted RDF data or the creation of a completely new dump. In summary, the ETL approach is not sufficiently flexible for very large and frequently changing data. It seems preferable to establish virtual RDF views over the existing relational database. While other tools, such as *D2R* and *Virtuoso RDF views*, offer mappings from relational databases to RDF, they cannot be easily applied to OpenStreetMap. The reason is that OpenStreetMap, like many other crowdsourced data sources, store much structure employing key-value pairs (or tags), instead of directly encoding this structure in the schema of the underlying relational database. This is commonly found in crowdsourcing applications so as to flexibly extend the structure of the data on the go. An RDF view mechanism for crowdsourced data will be able to handle this kind of data representation. Another reason why other relational database mappers could not be applied is their limited scalability. We will discuss this issue in greater detail in the evaluation section.

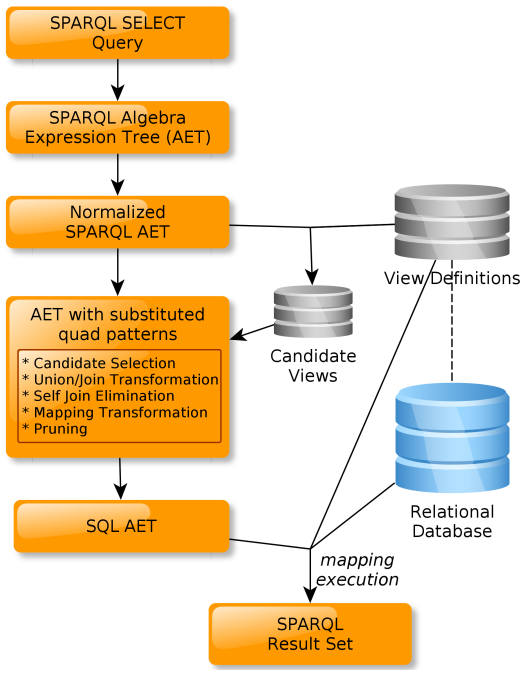

Fig. 3: The Sparqlify concepts and query rewriting workflow.

Figure 3 shows the query rewriting workflow in Sparqlify. The rationale of Sparqlify is to leave the schema of the underlying relational database schema unmodified and define RDF views over it. SPARQL queries can then be written against those views, which are expressed in *Sparqlify-ML* (mapping language) as defined in the next section. Sparqlify-ML is easy to learn for users, who are experienced in SPARQL and SQL and more compact than other syntactic variants such as R2RML. The left part of Figure 3 shows all steps, which are performed to answer a query. First, the query is converted into an algebra expression. This expression is subsequently converted to a normal form. Given the query patterns, relevant Sparqlify-ML views need to be detected. After this is done, the algebra expression is rewritten to include those relevant views. In a next step, optimisations on the algebra expression are performed to improve efficiency. Finally, this algebra expression can be transformed to an SQL algebra expression. For accomplishing this, we define a general relational algebra for RDB-to-RDF mappings. The SQL query, which was obtained, is executed against the relational database. Using the defined mappings, the SQL result set returned by the relational database can be converted to a SPARQL result set.

All of the above steps are explained in detail throughout the next sections.Our main contribution is a formalization, which goes beyond previous work by being capable to push the complete query execution using a single SQL query into the DBMS. As we demonstrate, our formalization and its Sparqlify implementation scales to larger databases and more complex queries than previous systems. Furthermore, we argue that Sparqlify has a low learning curve and high flexibility, i.e. support for deriving schema elements from rows in database tables and the addition of annotations.

5 Sparqlify-ML

In this section, we give an introduction to the Sparqlify-ML syntax. The innovation of Sparqlify-ML's view definition syntax is to blend the concepts of traditional SQL *Create View* statements with those of SPARQL *Construct* queries. The mapping language is therefore composed of features that a person working on RDB-RDF data integration is likely to be already familiar with. Limitations and future work of the language are discussed in Section 13.

Figure 4 juxtaposes the syntactic outline of the language with an example. The core syntax of a view definition is comprised of four parts:

- The *name* of the view.
- A *construct* clause, which consists of triple patterns which can be optionally associated with a specific named graph by surrounding them with `GRAPH` G { ... }, where G can be a variable name or an RDF literal. The syntax is therefore a slight extension of the *ConstructTemplate* production rule[10] of SPARQL.
- A *FROM* clause, where a *logical table* in the database can be specified. This can be either an SQL SELECT statement, the name of a physical table or the name of a view. The former needs to be escaped in double brackets, i.e. `[[SELECT ...]]`.
- The *variable definition clause* acts as the bridge between the RDF and SQL data models. It consists of a set of *variable definition statements* of the form $?var = termctor(expr_0, \ldots, expr_n)$, with:
  - *termctor* refers to the RDF term constructor, which can be `blankNode`, `uri`, `plainLiteral`, or `typedLiteral`.
  - $expr_i$ refers to expressions over sets of SQL column references of the logical table and constants. Syntactically, the currently supported operators in variable definition expressions are: `concat`, `urlencode` and `urldecode`.
  - *var* is a SPARQL variable being defined, which should be referenced from the construct clause.
- A *CONSTRAINT* clause that enables one to hint contstraints about variables on the RDF level and as such has no direct influence on the virtual RDF relation, but rather on query performance. The example in Figure 4 shows, that solely based on the definition of $?s = uri(?website)$ we have no information about the set of URIs being created. Specifying constraints enables Sparqlify, for instance, to detect and prune joins whose join condition equates variables with disjoint sets of prefixes. Syntactically, up to now only the prefix constraint can be explicitely specified. Internally, however, constraints are used to restrict predicates to URIs or subjects to blank nodes and URIs.

It is noteworthy that the 'variables' used as arguments of the term constructors are column references to the logical table and as such carry an SQL datatype.

Comparison of Sparqlify-ML with R2RML

In this section, we provide a brief comparison between two RDB-RDF mapping syntaxes. R2RML is a RDF vocabulary for expressing RDB-RDF mappings, originally inspired by the D2RQ mapping vocabulary. R2RML and Sparqlify-ML are both very recent and independent

[10] `http://w3.org/TR/rdf-sparql-query/#sparqlGrammar`

Outline

```
CREATE VIEW <name> AS
  Construct {
    <GraphConstructTriples> }
  }
  WITH
    <VariableDefinitions>
  CONSTRAIN
    <ConstraintDeclarations>
  FROM
    <LogicalTable>
```

Example

```
CREATE VIEW hotels AS
  CONSTRUCT {
    ?s a ex:Hotel .
    ?s rdfs:label ?l .
    ?s ex:vacancy ?v .
  }
  WITH
    ?s = uri(?website)
    ?l = plainLiteral(?name, 'en')
    ?v = typedLiteral(?vacancy,
              xsd:boolean)
  CONSTRAIN
    ?s prefix "http://ex.org/"
  FROM
    [[SELECT website, name, vacancy
      FROM hotels]]
```

Fig. 4: Sparqlify-ML view definition syntax.

efforts whereas the latter has since then evolved into a W3C Proposed Recommendation[11] The core concepts of R2RML have semantically equivalent counterparts in Sparqlify-ML and vice versa. However, users can benefit from Sparqlify-ML for two reasons:

- We expect Sparqlify-ML to be easier to learn and use for typical users than other mapping approaches, including R2RML, since it composes SPARQL and SQL constructs with few additional constructs.
- Sparqlify's syntax is more compact due to less syntactic noise created by using an XML based format.

A potential further advantage is that the syntax and the underlying formal model are conceptually closer in Sparqlify than in R2RML. The core concept underlying both syntaxes is the row wise RDF generation. We point out the most important similarities:

RDF Term Generation: R2RML defines the *term map* as a function that generates RDF terms from rows. These term maps can either be constant, column based or derived from a column using a template. Term maps are therefore conceptually equivalent to Sparqlify-ML's term constructors.

Quad Generation: For generating quads, R2RML utilizes *triple map*s and *graph map*s. Triple maps align *term map*s to form triples, whereas graph maps relate these triples to a named graph. As such, this is exactly what Sparqlify-ML expresses using a conventional SPARQL CONSTRUCT template.

Figure 5 shows a side-by-side comparison of the mapping languages for a specific example. Both syntactic formats can be converted to each other.

[11] `http://www.w3.org/TR/r2rml/`

Sparqlify-ML

```
Create View hotels As

  Construct {
    ?s a ex:Hotel .
    ?s rdfs:label ?l .
    ?s ex:vacancy ?v .
  }

  With
    ?s = uri(?website)
    ?l = plainLiteral(?name,'en')
    ?v = typedLiteral(?vacancy,
              xsd:boolean)

  From
    [[SELECT website, name,
      vacancy FROM hotels]]
```

R2RML

```
<HotelMap>
  rr:logicalTable [ rr:sqlQuery
  """  SELECT website, name,
  vacancy  FROM hotels """ ];

  rr:subjectMap [
    rr:column "website";
  ];
  rr:predicateObjectMap [
    rr:predicate rdfs:label;
    rr:objectMap
      [ rr:column "name";
     rr:language "en"];
  ];
  rr:predicateObjectMap [
    rr:predicate ex:vacancy;
    rr:objectMap
      [ rr:column "vacancy";
       rr:datatype
     xsd:boolean ];
  ].
```

Fig. 5: A simple view in Sparqlify-ML and R2RML.

6 SPARQL-SQL Rewriting

In this section, we describe the foundations for our SPARQL-SQL rewriting process, which focuses on scalability in terms of size of the relational databases as well as complexity of queries. We first provide a comprehensive formalization of the RDB-RDF mapping domain, especially the notions of RDB-RDF *mappings* and *view definitions*, whereas a set of view definitions spans the virtual RDF relation on which queries are answered. In the following section, we explain our approach to *view candidate selection*, which is the process of determining the potentially appropriate views given the quad patterns of a SPARQL query. Furthermore, we show how SPARQL queries are rewritten to refer to appropriate views by means of algebraic transformations. Subsequently, we introduce formal definitions for the adoption of SPARQL operators *join*, *left join* and *union* to mappings. Finally, we explain how the result sets for the different SPARQL query forms are constructed.

6.1 Formalization of View Definitions

This section starts with the introduction of several formal definitions that are fundamental to the Sparqlify system and RDB-RDF mapping in general. We proceed by first explaining (a) the nature of Sparqlify-ML's variable definitions, then explaining (b) the notion of a mapping, which expresses the transformation of a relation into a SPARQL result set, and finally we introduce (c) the view definition, which associates a mapping with a set of quad patterns, thereby yielding the virtual RDF graph on which queries are executed.

Definition 3 (Variable Definition) A variable definition is a triplet (v, tce, r), with v being the variable defined by the term constructor expression tce. Additionally, a set of *restrictions* r may apply for the variable.

For example, the definition of $?s$ in Figure 4 corresponds to

$(?s, uri(?website), uriPrefix(?s, \text{'http://ex.org/'}))$

Within a view definition, it is valid for a single variable to be defined multiple times: In this case the definitions are treated as *alternatives*: It is assumed, that for each row of the logical table, there exists *at most one* variable definition where *none* of the values in the referenced columns are NULL. For each row, where no such variable definition applies, the variable becomes unbound. Multiple definitions per variable occur during rewrites of SPARQL union operators, in which several members of the union provide values for the same SPARQL variable using different columns.

The Sparqlify RDB-RDF rewriting process has to handle *expressions* on three different levels: The SPARQL level, the SQL level and the mapping (short: map) level. Expressions can contain a large variety of operators, such as + (plus), - (minus) etc., which are usually dervied from XPath, SPARQL specific operators or SQL functions. We use the notions Θ_{rdf}, Θ_{map} and Θ_{sql}, respectively, to distinguish between the different formal interpretations that apply to an operator Θ across these levels.

- On the SPARQL level, we are dealing with expressions whose semantics are those of the SPARQL algebra.
- The map level mediates between SPARQL and SQL expressions. Map expressions are essentially an extension of SQL expressions, and therefore composed of column references, SQL literals and SQL functions. However, the distinguishing feature is the introduction of SPARQL's *type error* concept:
  Every SQL function label and SQL operator is associated with a declaration of its signature, i.e. the acceptable set of parameter type lists. If, for any function label or operator, in a map expression there exists no candidate, for which all the datatypes of the arguments are *compatible* to those of the parameters, the expression evaluates to *type error*. For example, a map expression $id =_{map} date$ with integer and string types, where $compatible(integer, date)$ does not hold for $=_{map}$, results in a type error. Note that our notion of datatype *compatibility* leaves the specification to the RDB-RDF mapping system implementation. The logical junctors $\wedge_{map}$, $\vee_{map}$ and $\neg_{map}$ are extended to ternary logic in order to treat type errors appropriately, as shown in Figure 6.
  Because map expressions follow semantics that are not supported by conventional relational database systems they need to be *pre-evaluated* in the RDB-RDF mapping system, before they are converted to SQL expressions that become part of a SPARQL query's corresponding SQL query.
- The SQL level captures conventional SQL expressions that are used in the relational algebra.

Evaluation in Ternary Logic Evaluation of SPARQL predicates follows a ternary logic composed of true (1), false (0) and type-error (e). When translating expressions of type *type-error* to SQL, they are treated as *FALSE*.

$\wedge_{map}(a, b)$

	1	**0**	**e**
1	1	0	e
0	0	0	0
e	e	0	e

$\vee_{map}(a, b)$

	1	**0**	**e**
1	1	1	1
0	1	0	e
e	1	e	e

$\neg_{map}(a)$

1	0
0	1
e	e

Fig. 6: Ternary Logic Truth Tables used for the treatment of type errors.

Definition 4 (RDF Term Constructors) An RDF term constructor is a *function* that takes the four arguments $(type, value, datatypeTag, languageTag)$. Each of its arguments are map expressions that upon evaluation yield SQL literals, which must conform to the following restrictions:

- *type* is an integer in the range [0..4], with 0 = *blankNode*, 1 = *uri*, 2 = *plainLiteral* and 3 = *typedLiteral*. Note that sorting by the type field yields an order consistent with the SPARQL standard's ORDER BY clause[12].
- *value* can have arbitrary datatype. The *physical* datatype of the underlying SQL relation is retained and associated with the value.
- *languageTag* is a string, whose value corresponds to a language tag.
- *datatypeTag* is a string, which indicates a *logical* RDF datatype. This field does not have any effect on the rewriting, but is considered when transforming a SQL result set into an RDF relation. The semantics of the RDF datatype referenced by the datatype tag should be similar, ideally equal, to those of the value's datatype, such mapping PostgreSQL's `text` datatype to `xsd:string`.

[12] `http://w3.org/TR/rdf-sparql-query/#modOrderBy`

A principle, which we see as fundamental for efficient RDB-RDF mapping approaches, is that during rewriting, RDF term constructor expressions preserve the semantics of their value's SQL datatype. For instance, if an RDF literal is created from a float value, then operators, such as >, <, =, have numeric semantics. This has a severe implication: Operations on RDF terms are implicitly already operations on SQL literals. For this reason, RDF datatype tags used in view definitions should be aligned with those of the underlying columns, otherwise Sparqlify will not behave according to the SPARQL semantics because of this discrepancy.

Only when evaluating the rdf-term-constructor expressions it is required for the *atomized* value field to lie within the lexical space. Otherwise, Sparqlify does not impose this restriction.

The term constructors for blank nodes, URIs and literals are specializations of $rdfTerm$:

- $blankNode(e) \rightarrow rdfTerm(0, e, '', '')$
- $uri(e) \rightarrow rdfTerm(1, e, '', '')$
- $plainLiteral(e) \rightarrow rdfTerm(2, e, '', '')$
- $plainLiteral(e, l) \rightarrow rdfTerm(2, e, '', l)$
- $typedLiteral(e, d) \rightarrow rdfTerm(3, e, d, '')$

For this representation, we define the following rewrite rules that translate SPARQL (in)equality operators into corresponding map operators, given two rdfTerm expressions a and b. We use the notion a_i with i in the range [1..4] to refer to an rdfTerm's correpsonding argument, i.e. 1=type, 2=value, 3=datatypeTag, 4=languageTag.

- $a =_{rdf} b \rightarrow a_i =_{map} b_i$ (component wise equality)
- $a <_{rdf} b \rightarrow a_i <_{map} b_i$

The rules can analogously defined for <=, >, >= and are omitted here for brevity.

Definition 5 (Atomization of SQL literals) The following function yields, for a given SQL literal and datatypeTag, a corresponding representation in the lexical space of the datatypeTag:
$atomize\,(SQLliteral, datatypeTag) \rightarrow String$.

This concept enables the support of arbitrary datatypes, such as spatial ones. For example, PostGIS introduces a special geometry datatype, for which we define

$atomize$(geometry-SQL-literal, ogc:WKTLiteral)

to yield a WKT[13] representation of the value.

An RDB2RDF mapping now declaratively expresses the conversion of a relational result set into a corresponding SPARQL result set.

[13] Well-known text: `http://www.opengeospatial.org/standards/ct`

Definition 6 (RDB-RDF Mapping) An RDB-RDF *mapping* is a tuple (D, l), with D a set of variable definitions and l being a logical table, represented as a relational algebra expression. We denote the set of all mappings as $\mathcal{M}$.

Definition 7 (View Definition) An RDF view definition is a tuple (T, m), where the template T is a set of quad patterns, and m an RDB-RDF mapping as defined above. It is assumed that $vars(T) = vars(m)$, i.e. all defined variables are referenced in the pattern, and no variable in the pattern is undefined.

These definitions cover the central concepts for mapping tabular data to RDF. In fact, these concepts are already sufficient for being applied in the context of creating RDF dumps from relational data. Given a view definition, which refers to a relation, we can create RDF data as follows: For each row of the relation, evaluate the view's variable definitions, and use the obtained mapping from variables to RDF terms in order to instantiate the template of the view. The set of all triples (quads without the graph component) obtained by this forms the RDF dump. However, in order to use the introduced concepts for SPARQL-SQL rewriting, we require additional concepts:

Definition 8 (Binding and Binding Function) A *binding* b is a subset of $\mathcal{V} \times \mathcal{P}(\mathcal{V} \cup \mathcal{L})$, where $\mathcal{P}$ denotes the powerset constructor. We denote the set of all bindings as $\mathcal{B}$. Furthermore, we require for every b the existence of a partial function

$$b_f(k) : \left\{ \begin{array}{ll} v & (k, v) \in b \\ undefined & \text{otherwise} \end{array} \right\}$$

We refer to b_f as the *binding function* of b.

Note that this implies, that a binding can only associate at most a single v with any k. In our rewriting approach, bindings are used to capture the relation of variables of a SPARQL query to those used in a views' template.

In the following, we *always* use bindings to map from *query variables* to *view variables* and literals. Furthermore, we define helper functions for operations on sets of variables:

- $keys(b, v) := \{k | \exists k : v \in b(k)\}$ is the function that for a given variable (or constant) v returns all the binding's keys that map to sets containing v.
- $Keys(b, V) := \bigcup_{v \in V} \{keys(b, v)\}$ is similar to above, but takes a set of variables V as argument.
- $Values(b, K) := \bigcup_{k \in K} \{b(k)\}$ returns the union of the value-sets for a set of keys K.

Definition 9 (Binding Merge) Two bindings b_1 and b_2 can be merged by combining all their mappings:

$$b_1 \oplus b_2 := \{q \rightarrow v | q \in dom(b_1) \cup dom(b_2), v = b_1(q) \cup b_2(q)\}$$

Definition 10 (Creation of Bindings) In our approach, bindings are always created from a pair of quad patterns. We define β as the function which creates a binding by mapping the respective components of two quad patterns r and s:

$$\beta(r, s) = \bigoplus_{i \in I} \{r_i \rightarrow \{s_i\}\}$$

Definition 11 (Binding Closure) We use b^* to denote the binding closure of a binding $b = b^0$, which is computed by iteratively calculating b^{i+1} until a fixpoint is reached:

$$b^{i+1} := \bigcup_{k \in dom(b^i)} \left\{k \rightarrow Values(b^i, Keys(b^i, b^i(k)))\right\}$$

The binding closure is used to establish a mapping from a query variable to *all* its related view variables. Because view variables are associated with a set of constraints, the binding closure indirectly gives access to all constraints that must simultaneously apply to a query variable. An example is shown in Figure 13.

Definition 12 (View Instance) A *view instance* is a pair (v, b), whereas v is a view definition and b is a binding as defined above.

Such view instance constructs are used for binding the variables of a query to those of a certain view.

7 Candidate View Selection

A SPARQL query is composed of graph patterns, whereas the most primitive one is the quad pattern. In this section we show how, after some pre-processing, for each quad pattern of a query the corresponding set of candidate view instances is be retrieved. We demonstrate each of these steps based on the example query and the view definitions shown in Figure 7. Additionally we explain how to derive a new query which references these candidate views and translates it into a mapping, i.e. a pair comprised of an SQL algebra expression and a set of variable definitions. Sparqlify's approach for candidate view selection is inspired by [11], which discusses RDF views on RDF data. Our contribution in this area is the adoption for relational data, which covers slightly different formalizations and new optimizations.

Given a SPARQL query *string* subject to rewriting, the pre-processing consists of parsing the query, translating it into a SPARQL algebra expression, and normalizing it. Normalization means replacing blank nodes with variables and replacing constants (URIs and literals) with newly allocated variables, which are constrained to the constant values through the introduction of appropriate filter operators (cf. Figure 8).

Original Query	*Constants Normalized*

`Project ((?s ?o), {`
`  QuadPatternSet({`
`  ?s ?t ?s .`
`  ?s ex:interest ?o`
`  })`
`})`
	`Project ((?s ?o), {`
	`  Filter({v1 = ex:interest},`
	`  QuadPatternSet({`
	`  ?s ?t ?s .`
	`  ?s ?v1 ?o`
	`  })`
	`}) )`

Fig. 8: Replacing constants in quad pattern sets with newly allocated variables which are constrained by appropriate filter operators.

This normalization offers two advantages: First, it does not matter whether a constant appears in a quad pattern or whether a variable is equated to a constant in a filter clause. We only have to deal with the latter case. Second, the use of variables enables us to associate constraints with them: Naturally, the original constant becomes one of its constraints. However, in the process of binding the query variable to view variables, we are now also able to keep track of constraints inherited through the query-view binding. Thus we can, for instance, check whether a variable's set of constraints is inconsistent, and perform optimizations accordingly.

7.1 View Candidate Detection Algorithm

The basic approach to candidate view selection is to perform a bottom-up traversal of a normalized SPARQL algebra expression, and derive a new one where each of the quad patterns has been replaced with an appropriate *union of view instances*. QuadPatternSets can be seen as nested joins on its constituent quad patterns, which consequently translates to *joins of unions of view instances*. Subsequently, any joins of unions are transformed into unions of joins (cf. Figure 9), which enables the elimination of self joins. Finally, the view instances are transformed into mappings, and the query is translated according to the algebraic transformations defined in Section 8.

The inputs of the view candidate discovery algorithm are a quad pattern q, a set of views V and a constraint expression e. In our approach, we derive the expression e from the filter conditions that apply for a certain quad pattern. e can therefore be assumed to be expressed in terms of the variables of the SPARQL query. Note that we keep the concepts of selecting candidates for which a constraint e holds distinct from how to obtain e. For example, in our normalized query, the

Query

```
SELECT ?s ?o {
  ?s ?t ?s .
  ?s ex:interest ?o .
}
```

Result Set	
s	**o**
ex:John	"Geography"

Nominations View

```
CREATE VIEW nominations AS
  CONSTRUCT {
    ?x ex:nominated ?y .
  } WITH
    ?x = uri(?ns)
    ?y = uri(?no)
  FROM
    sql_nominations
```

sql_nominations	
ns	**no**
ex:John	ex:John
ex:Anne	ex:Lizy

Interests View

```
CREATE VIEW interests AS
  CONSTRUCT {
     ?h ex:interest ?i .
  } WITH
     ?h = uri(?is)
     ?i = plainLiteral(?io)
  FROM
    sql_interests
```

sql_interests	
is	**io**
ex:John	"Geography"
ex:Anne	"Quantum Physics"

Fig. 7: A query and its expected result set when executed on the two given views and corresponding relations. The query asks for the interests of people that nominated themselves. This example demonstrates the case of a self-reference. (Relations are assumed to hold full URIs, which were abbreviated here.)

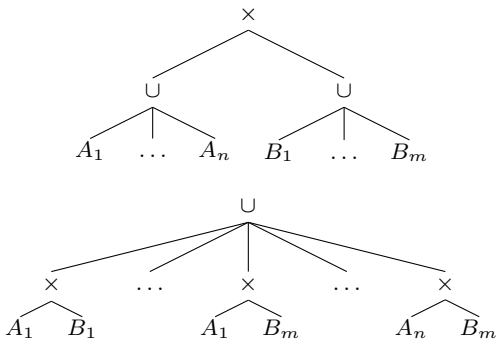

Fig. 9: Transformation of unions of joins into joins of unions within a part of an algebra expression. In our setting, A_i and B_i correspond to individual candidate view instances of two quad patterns A and B.

constraint v_1 = ex:interest holds for the quad pattern `(:g ?s ?v1 ?o)`.

Our approach for finding the candidate view instances $cand(q, V, E)$ is to compute, for every q, all possible bindings to the quads of the views, and only retain those for which a (partial) evaluation of e does not yield $FALSE$, formally:

$$cand(q, V, e) := \bigcup_{v \in V} \{(v, b) | r \in (v_T), b = \beta(q, r) and \quad (1)$$
$$\neg(partialeval(e, \downarrow_b (v_{M_D}) = FALSE)\}$$

- The operator $partialeval(e, D)$ evaluates the expression e in regard to the variable definitions D. If D is inconsistent, it yields false.
- The operator $\downarrow_b (v_{M_D})$ denotes the set of variable definitions obtained by expanding the original variable definitions v_{M_D} with the binding b. It is formally defined in Subsection 8.1.

Table 2 shows the bindings obtained for our running example. The query quad (?s ?v1 ?o) only has a single non-inconsistent view-instance candidate, whereas (?s ?t ?s) has two such candidates.

A naive candidate selection approach may chose candidates unconditionally (i.e. $e = TRUE$) but this leads to an exponential growth in the number of rewrites. For example a quad pattern set with *members* for which there exist 10 candidates each would result without further optimization in a union of joins with 10^5 members. In section Subsection 9.1 we explain our implementation and optimizations of the candidate selection process.

7.2 Elimination of Self Joins

A *self join* situation occurs, if a set of view instances sharing the same view is equivalent to a single view instance (using the same view) using an appropriate binding. In other words: We are interested in detecting cases, where multiple bindings to the same view can be replaced by a single one. An example is shown in Figure 10.

Given a view instance set, we can detect and eliminate self-joins using the algorithm in Figure 11.

In the following section we introduce a relational algebra for mappings, which serves two purposes: Primarily, it enables the translation of a SPARQL algebra expressions whose leaves are mappings into a single mapping where all operations have been pushed into the SQL algebra. Secondarily, it allows the reuse of traditional SQL optmizations.

8 A Relational Algebra for RDB RDF Mappings

In this section we first define the SPARQL 1.0 operators *rename*, *join*, *left join* and *union* for mappings.

View	View Quad	Bindings for the query quads ?s ?t ?s	?s ?v1 ?o
types	?x ex:nominated ?y	?s = {?x, ?y} ?t = {ex:nominated}	?s = {?x} ?v1 = {ex:nominated} (X) ?o = {?y}
comment	?h ex:interest ?i	?s = {?h, ?i} ?t = {ex:interest}	?s = {?h} ?v1 = {ex:interest} ?o = {?i}

Table 2: View instances for the two query quads. The binding marked with (X) is inconsistent with the constraint "?v1 = ex:interest" which is imposed by the query.

Query	*View*	Binding and inverse binding for the two view quads	

SELECT * {
?s ?t ?s .
?s ?v ?o .
 	
FILTER(?v = ...) .
}
	Create View types As
	Construct {
	?x ex:nominated ?x .
	?x ?c ?z .
	} With
	...
		?s -> {?x}
		?t -> {ex:nominated}
		---------- 	
		?s -> {?x, ?z}
		?v -> {?a}
			?x -> {?s}
			---------- 	
			?x -> {?s}
			?z -> {?s}
			?a -> {?v}

Fig. 10: Example for a self-join: A query against the given view can be answered directly by the view using an appropriate binding. There is no need for joins.

```
function checkMerge(viewInstanceA, viewInstanceB) {
  if(viewInstanceA.view != viewInstanceB.view) { return false; }

  // A binding maps query variables to sets of view variables
  // Get the inverse bindings, thus view-variable -> query variables
  inverseA = viewInstanceA.binding.inverse;
  inverseB = viewInstanceB.binding.inverse;

  // Now check if each parent variable in inverseA maps to the same
  // query variables as in inverseB
  // If that is the case, we have a self join
  for(viewVarA in inverseA.keys) {
    queryVarsA = inverseA[viewVarA];
    queryVarsB = inverseB[viewVarA];

    if(queryVarsB.isEmpty) { continue; }

    intersection = queryVarsA intersect queryVarsB;

    // If the inverse bindings map one of the view variables
    // to different query variable we cannot merge
    if(intersection.isEmpty) { return false; }
  }

  return true;
}
```

Fig. 11: Pseudo-code algorithm for checking whether bindings can be merged in order to remove self joins.

8.1 Deriving Mappings from View Instances

The leaf nodes of a SPARQL query that was rewritten on a set of views are formed by view instances, i.e. (View, Binding) pairs, whereas a view is a pair (Template, Mapping). This enables us to create a pair (Mapping, Binding), which we refer to as *mapping instance*.

$\downarrow: (\mathcal{M}, \mathcal{B}) \rightarrow \mathcal{M}$ that creates a new mapping from an existing mapping instance by expressing its variable definitions in terms of the query variables of a binding. Recall that a binding may associate a single query variable with multiple view variables. In this case, all affected view variables must be constrained to be pair-wise equal because the query variable cannot be bound to multiple values at once. As a consequence, after equating all view variables, we can pick an arbitrary one of them, and create a new variable definition with the query variable and the picked variable's defining expression.

Formally: Given a mapping m and a binding b. Let D be the variable definitions, and l the logical table expression of m. We define the function $\downarrow(m, b)$, which derives a mapping m' from a mapping instance (m, b) as follows:

$$\downarrow(m, b) = \begin{pmatrix} pick(D), \\ \sigma_{translate_{sql}(\downarrow_D(D,b)(l))} \end{pmatrix}$$

Note that in cases, where a query variable maps to multiple view variables, we need to create a selection on the logical table l.

$pick(D, b)$ is a function, that for every query variable picks the variable definitions for only one of the view variables it is related to via the binding:

$$pick(D, b) = \bigcup_{k \in \text{dom}(b)} pick_d(D, b, k)$$

The function $pick_d(D, b, k)$ picks the definitions for one of a query variable's related view variables (denoted as vv in the following formula).

$$pick_d(D, b, k) = \{(k, D(vv)) | vv \in b(k))\}$$

Analogously, we define $\downarrow_d (D, b)$, which equates all of the variable definitions a query variable is indirectly related to via a binding:

$$\downarrow(D, b) = \bigwedge_{k \in \text{dom}(b)} \downarrow_d (D, b, k)$$

With its per-query variable pendant:

$$\downarrow_d (D, b, k) = \text{equals}_{rdf}\,(D, b(k))$$

The helper function $equals_{rdf}(D, V)$ yields a conjunctive normal form composed of equality expressions for a set of definitions D and a set of variables V:

$$equals_{rdf}(D, V) =$$

$$\begin{cases} false & \text{if } |V| = 0 \\ true & \text{if } |V| = 1 \\ \bigwedge_{v \in V} \bigvee_{e_i, e_j \in D(v), e_i \neq e_j} \{e_i =_{rdf} e_j\} & \text{otherwise} \end{cases}$$

In the formula, D maps variables to definition expressions, such as $concat(\ldots)$. This means, the definitions are not predicate expressions and, therefore, do not express a constraint. Therefore, if $|V| = 1$, the result is $true$. However, if V contains at least 2 variables, then all corresponding definitions in D can be equated. Note that a set of definitions for certain variables are seen as alternative, and therefore result in disjunctions.

8.2 Rename

Given a mapping $R := (R_D, R_l)$, we use the notation $\rho_{[a/b]}(R) := \left(\rho_{[a/b]}(R_D), \rho_{[a/b]}(R_l)\right)$ to replace all column references to b with a.

8.3 Natural Join

The natural join between two mappings R and S is based on their commonly defined variables The natural join involves renaming of the columns in S, constructing an appropriate join condition based on the variable definitions, and creating a new variable definition . We define $R \times S$ as follows:

- First, we need to ensure that the column names of R and S are disjoint. Let $c_1, \ldots, c_n$ be the column names common to R and S, we derive $S' = \rho_{[x_1, \ldots, x_n / c_1, \ldots, c_n]}(S)$ with $x_i, \ldots, x_n$ being a new distinct column names neither in R nor S.
- The join condition is construced as follows: Let H be the set of common defined variables in R and S. Further, let R_{D_d} denote the set of alternative definitions for d in R, with $d \in D$. The join condition jc on RDF level is the following CNF:
  $jc(R, S) := \bigwedge_{d \in D} \left\{ \bigvee_{a \in R_{VD_d}, b \in S_{VD_d}} \{d_a =_{rdf} d_b\} \right\}$
- The new set of variable definitions are all those of S together with only those of R, where the variables were not common to both.

Combination	Result	Inconsistent?
$A_1 \times B_1$	?s = {?h, ?x, ?y} ?t = {ex:nominated} ?v1 = {ex:interest} ?o = {?i}	no
$A_2 \times B_1$	?s = {?h, ?i} ?t = {ex:interest} ?v1 = {ex:interest} ?o = {?i}	no

Table 4: The result of joining the candidate view instances.

$$R \times S \rightarrow (\sigma_{\text{translate\_sql}(jc(R,S'))}\,(R \times S'))$$

As an optimization, the final step is to prune all elements of $\mathfrak{C}$ that yield unsatisfiable join conditions, this is shown in Table 4. If this step was not applied, the rewritten SQL query would contain JOINS that yield empty result sets and are therefore unnecessary.

8.4 Union

A union of two or more mappings, $Union(m_1, m_2, \ldots, m_n)$ is conceptually similar to the SQL outer union operator.

A noteworthy aspect about unions is, that they can lead to multiple bindings for the same variable: This case arises when two or more of its members define variable bindings for the same variable, such as one member with $?s = uri(?workpage)$ and another with $?s = uri(?age)$. Naturally, each of these bindings is only valid for the subset of the union's result set, which was contributed by the respective member. In order to disambiguate which bindings to apply, we can exploit the property of having taken the outer union. This ensures that all columns not originally belonging to a member will be filled with NULLs. Therefore, given a tuple of the union, any binding for which any of its referenced columns is NULL does not apply. Note that in the case of LEFT-JOINs it may happen that none of the bindings apply, which results in an unbound RDF term. Special treatment needs to be done with bindings to constants, such as $?o = typedLiteral(\text{“5”}, xsd : double)$: In order to discriminate for which member they apply, they are pushed into columns of appropriate data type into the respective member-relation. The binding is then updated accordingly, such as $?o = plainLiteral(?genColOfTypeDouble, xsd : double)$.

One reasonable optimization is to reduce the number of columns a full outer union would generate. Sparqlify achieves this by factoring out structural equivalent binding expressions and assigning them the same set of columns (see Figure 12).

?s ?t ?s		?s ?v1 ?o
A_1: (types, `?s = {?x, ?y}` `?t = {ex:nominated}`)		
	×	B_1: (interests, `?s = {?h}` `?v1 = {ex:nominated}` `?o = {?i}`)
A_2: (interests, `?s = {?h, ?i}` `?t = {ex:interest}`)		

Table 3: Joining the obtained sets of candidate view instances. Every non-inconsistent combinations of all candidates results is considered a join of the candidates. The set of all non-inconsistent combinations is eventually translated into a union. Note that every combination can be checked for satisfiability after merging all the bindings.

Nominations' Mapping

```
?s = {uri(?ns)}
?p = {ex:nominated}
?o = {uri(?no)}
```

sql_types	
ns	**no**
ex:John	ex:John
ex:Anne	ex:Lizy

Interests's Mapping

```
?s = {uri(?is)}
?p = {ex:interest}
?o = {plainLiteral(?io)}
```

sql_comments	
cs	**co**
ex:John	"Geography"
ex:Anne	"Quantum Physics"

Union's Mapping

```
?s = {uri(?s)}
?p = {uri(?p)}
?o = {uri(?o_1), plainLiteral(?o_2)}
```

union			
s	**o_1**	**o_2**	**p**
ex:John	ex:John		ex:nominated
ex:Anne	ex:Anne		ex:nominated
ex:John		"Geography"	ex:interest
ex:Anne		"Quantum Physics"	ex:interest

Fig. 12: The resulting mapping after optimizing the outer union yield by processing the query `SELECT * { ?s ?p ?o }` on the given views. Because all variable definitions of `?s` were structurally equivalent, they could be combined into the same column. The predicate URIs had to be pushed into columns (of type string) into the respective member as to discriminate them. The corresponding columns could then be collapsed as well because of structurally equivalent variable definitions. The object columns could not be further optimized. Note that the column names of the projection are usually automatically assigned by the system. Here we have labeled them for clarity.

8.5 Slice

The slice operator is a combination of SQL's limit and offset. It translates directly into a mapping's logical table algebra expression:

$$slice_{rdf}^{limit,offset}((D, L)) = (D, slice_{sql}^{limit,offset}(L))$$

9 Implementation

9.1 Optimized Candidate Selection

The challenge with Equation 1 is to *efficiently* find the set of candidate view instances for a given query quad p. Our solution is based on deriving *restrictions* from the *constraints* that apply to the query and view variables.

Note that the quads we are dealing with only consist of variables due to the constant normalization and recall that constraints are predicate expressions (which can be *evaluated*), whereas restrictions are functions that map variables to objects (and correspond to *descriptions* which are used in optimizations). Currently, Sparqlify supports the following two restrictions:

- *rdf-term-type restrictions:*
  Let $T = \{blank, uri, plainLiteral, typedLiteral\}$ be the set of RDF term types. We define a term type restriction function as $\mathfrak{t} : Var \rightarrow \mathcal{P}(T)$.
- *prefix restrictions:* This function maps variables to a set of URI prefix strings. $\mathfrak{n} : Var \rightarrow \mathcal{P}(String)$

In the future, this may be extended to include, e.g., equality restrictions $equals : \{?p \rightarrow ex : nominated\}$ or value range restrictions, for instance $rangeRestriction : \{?age \rightarrow [0, 100]\}$. Note that negated restrictions, such as "does not have certain namespaces" are currently unsupported.

Deriving the restrictions for the variables of view definitions is relatively simple:

- Based on the term constructor of a variable definition, we can directly infer the variable's RDF term type restriction.
- The prefix restrictions can be directly derived from a view's `startsWith` constraints. Note that a prefix constraint implies a URI-term-type restriction.
- As stated, `startsWith` constraints can be derived from URI term constructors whose argument either has the form *string-constant* or *concat(string-constant, ...)*. The restriction can then be derived in a separate step.

We use these restrictions for indexing the views, as shown in Table 5.

Deriving restrictions for a quad is more complex, as it requires analyzing all filter expressions that effect the quad in question.

Given a constraint expression c, we can infer the restrictions that apply to $vars(c)$ by:

1. If c is converted into conjunctive normal form (CNF), i.e. $\bigwedge_i \bigvee_j (\neg) x_{ij}$, then we can derive "global" restrictions based on all single-literal-clauses, i.e clauses of

view_index								
name	**quad**	**s_type**	**s_pre**	**p_type**	**p_pre**	**o_type**	**o_pre**	**o_lang**
types	?x ex:nominated ?y	uri		uri	rdfs:type	uri		
comments	?h ex:interest ?i	uri		uri	ex:interest	plainLiteral		en

Table 5: Index table for the views of the running example. The "pre"-columns contain prefix constants that apply to the components (g, s, p, o) of the corresponding view-quad. If a component in a quad has more than one prefix, the same view-quad may be indexed multiple times.

the form $\{L\}$. Note that the elements of the clauses are expressions. For instance, from a clause $\{equals\,(?p, ex:nominated)\}$ we can derive that the prefix of ?p must `be ex:nominated`.

2. If c is converted into disjunctive normal form (DNF), i.e. $\bigvee_i \bigwedge_j (\neg) x_{ij}$, then it is possible to derive constraints on a per-clause basis. We exploit this property to perform lookups of view candidates in the view index on a per-clause basis.

Once we have obtained a set of views V_q for a given query quad q, the next step is to create the view instances. Note that when binding the query variables to those of a view, the query variables inherit all constraints and restrictions that apply to the view variables. The binding closure thereby helps to capture indirect constraints: In Figure 13, we see that the closure additionally adds $\{?a \rightarrow ?y\}$ and $\{?c \rightarrow ?x\}$. Therefore any restrictions on $?y$ and $?x$ carry over to $?a$ and $?c$, respectively. These new restrictions can then be used to make successive lookups more selective.

Optimized View Candidate Selection Algorithm

Having described how to perform an efficient lookup for a single query quad in the previous section, we here present our optimized approach for finding candidate views for a quad pattern.

Our optimized algorithm for finding view candidates for a single QUAD-PATTERN-SET set proceeds as follows:

- Pick the quad that has the least number of consistent bindings. The rationale is, that as soon as a binding becomes inconsistent we can cut off searching this branch as any further JOIN will yield an empty result set.
- Order the views that share most variables first. As a result, the variables will inherit any constraints (from views and the query) early which gives a chance to quickly detect inconsistencies. If there are no more quads that share variables with the prior ones, pick the next one with the fewest candidates.
- By doing this, we have now established an order on the query quads. The next step is to perform the JOIN as of Equation 1 in this order. However, this time we take all restrictions into account, and reperform the lookups. Whenever restrictions become unsatisfiable, we can skip the building of the current join candidate and advance to the next one.
- For every candidate binding x, recursively perform the following:
  - If possible, take the next query-quad q, and perform a lookup for all its view candidates V_q by considering all corresponding new and prior constraints. For each candidate, create a new binding z, by merging the prior bindings x with the one of the new candidate. If this new binding z is inconsistent, try the next view instance candidate. Otherwise, continue with the next query quad.
  - If there is no next query quad, we have successfully created a set of candidate view instances whose bindings do not result in knowingly unsatisfiable join conditions. We can then add this set of instances to our result set.

Following the algorithm in Figure 7.1, we therefore start with the set of view instance candites of (?s ?v1 ?o). The result is depicted in Table 4.

9.2 The Sparqlify System

Overall, the Sparqlify system is comprised of:

- The Sparqlify *Engine*, which implements the SPARQL-SQL rewriting approach detailed in the next section.
- The Sparqlify *Server*, which provides an HTTP SPARQL interface.
- The Sparqlify *Platform*, which bundles the SPARQL explorer SNORQL[14] and the Linked Data Publishing tool Pubby[15]
- The Sparqlify Mapping Language (Sparqlify-ML), which offers similar expressivity as D2RQ-ML and R2RML. However, it is based on an adapted SPARQL grammar rather than RDF and therefore greatly reduces syntactic noise.

As common in other systems, Linked Data is served by issuing a SPARQL DESCRIBE query to the underlying

[14] `https://github.com/kurtjx/SNORQL`
[15] `http://www4.wiwiss.fu-berlin.de/pubby/`

Query	*View*	*Binding*	*Binding Closure*
	`Create View example As`		
`SELECT * {`	`  Construct {`		
`  ...`	`  ...`	`?a -> { ?x }`	`?a -> { ?x, ?y }`
`  ?a ?b ?b ?c`	`  ?x ?x ?y ?y`	`?b -> { ?x, ?y }`	`?b -> { ?x, ?y }`
`  ...`	`  ...`	`?c -> { ?y }`	`?c -> { ?x, ?y }`
`}`	`  }`		
	`  With ...`		

Fig. 13: Example of a binding of variable between a quad of the query and one of a view.

system – in our case the Sparqlify Engine. By providing Linked Data in various serialization formats and an HTTP SPARQL interface, the Sparqlify system covers essential Web data publication requirements.

10 Sparqlify in LinkedGeoData

Section 3 outlined the challenges encountered with the LinkedGeoData project and the previous sections described the formal foundations for scalable SPARQL-SQL rewriting systems. Here we present the application of Sparqlify in the context of the LinkedGeoData project.

An overview of the revised structure of LinkedGeoData and the role of Sparqlify is depicted in Figure 14. In contrast to the previous approach (see Figure 2), many components depend centrally on Sparqlify.

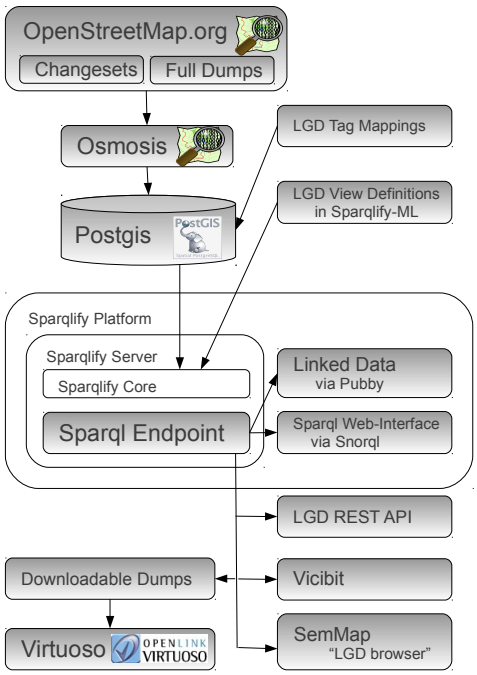

Fig. 14: Overview of the new LinkedGeoData architecture. Most components now uniformly depend on the SPARQL endpoint provided by Sparqlify.

10.1 Mapping OpenStreetMap to RDF

In this section, we explain our approach to mapping the OpenStreetMap database to RDF. The Osmosis tool ships with two schemas for the PostgreSQL database, namely *simple* and *snapshot*[16]. Both schemas are capable of storing the same information, however, the snapshot one uses the special *hstore datatype*[17], which enables the storage of sets of tags in a single row. Since this datatype defines a special set of operators, it is not yet supported by Sparqlify (and to the best of our knowledge neither by any other RDB-RDF mapping tool). For this reason we used the purely relational simple schema.

At the core, the LinkedGeoData database is comprised of:

- All tables from the simple schema. The essential tables are `nodes`, `ways` and `relations`. These tables have a primary key of type bigint, and additional columns for the metadata about the respective OSM entities, such as the user ID and the date of the most recent change. For each table there exists also a corresponding tag table with the structure ({way, node, relation}_id, k, v).
- Several *translation tables* that are used to store the mappings between OSM tags and RDF resources. In order to avoid confusion with the RDB-RDF mappings, we use the term "(tag) translation" to refer to the concept of translating OSM tags to RDF terms.
- A set of SQL helper views that hide underlying complexity. The complexity stems from joining the simple schema with the translation tables, and the use of function indexes on tags whose values can be converted to numeric values.
- Additional indexes for the keys and values of the OSM tags. Furthermore we use functional indexes

[16] `http://wiki.openstreetmap.org/wiki/Database_schema#Database_Schemas`

[17] `http://www.postgresql.org/docs/9.1/static/hstore.html`

to capture those tag-values that can be parsed as numerics.

Additionally, there are tables for the following data integrated from external sources:

- Our *SameAs links* to DBpedia and GeoNames.
- Icons from SJJB Management[18].
- Multilingual labels for the schema of the LinkedGeoData ontology obtained from the OSM translation projects coordinated on TranslateWiki[19].

Translation tables Whereas the subject of the triples generated from the tag tables is determined by an OSM entity's ID, their predicates and objects depend on the tags. The following tables are used in LGD to store the information that map tags(k, v) to RDF.

- lgd_map_datatype: Associates a key with a numeric datatype (integer or real).
- lgd_map_label: Associates $(k, v) with (label, languageTag)$
- lgd_map_literal: Maps k to a property, and transforms v into a literal with specified language tag.
- lgd_map_resource_k: Any entity having a key k is mapped to a pair of URIs for the property and object.
- lgd_map_resource_kv: Same as above, except that both k and v must exist.
- lgd_map_property: Covers cases, where v is already a URI, and only k needs to be mapped to a property. Used for keys such as *website* or *depiction*.

Use case Figure 16 shows a use case of LinkedGeoData for finding tram stops within a certain region[20]. The spatial predicate `ogc:intersects` and the function `ogc:geomFromText`[21] translate eventually to the PostGIS's functions `ST_Intersects` and `ST_GeomFromText`. The underlying view definitions's WITH clause only contains an entry

$?g = typedLiteral(?geom, ogc:WKTLiteral)$

for mapping the geometry type. The type is then handled appropriately by the Sparqlify engine. Due to the the mapping process with Sparqlify, LinkedGeoData now also exposes information about OSM relations, such as the administrative boundaries[22], or which tram stops belong to which tram routes. The full compatibility of the LinkedGeoData schema with the OSM simple one also shows, that Sparqlify fits nicely into the OSM tool chain.

[18] `http://www.sjjb.co.uk/mapicons/`

[19] `http://translatewiki.net`

[20] It can be run on http://test.linkedgeodata.org/snorql

[21] `Forconveniencewetiedthisfunctionintotheogcnamespace.`

[22] See for example `http://linkedgeodata.org/page/triplify/relation51477/members` for the administrative boundary of Germany.

```
Create View classes_kv As
    Construct {
        ?s a owl:Class .
        ?s lgdm:sourceTag ?t .
        ?t lgdm:key ?k .
        ?t lgdm:value ?v .
    }
    With
        ?s = uri(?object)
        ?t = uri(concat(?object, "/key/",
                 ?k, "/value/", ?v))
        ?k = plainLiteral(?k)
        ?v = plainLiteral(?v)
    Constrain
        ?s prefix "http://linkedgeodata.org/ontology/"
        ?t prefix "http://linkedgeodata.org/ontology/"
    From
        [[SELECT object, k, v FROM
         lgd_map_resource_kv WHERE
         property = 'rdf:type']]
```

lgd_map_resource_kv

k	v	property	object
amenity	university	rdf:type	lgdo:University
route	tram	rdf:type	lgdo:TramRoute
sport	skiing	lgdo:featuresSport	lgdo:Skiing
...			

Fig. 15: Sparqlify-ML and a table as being used in LinkedGeoData. The view maps tags to classes, and additionally exposes for each class the information from which tag it was created.

```
SELECT * WHERE {
  ?b a lgdo:TramStop .
  ?b rdfs:label ?l .
  ?b geom:geometry ?geom .
  ?geom ogc:asWKT ?geo .
  Filter(ogc:intersects(?geo,
    ogc:geomFromText('POLYGON((8 50, 12 50,
        12 55, 8 55, 8 50))')))
}
Limit 100
```

Fig. 16: A SPARQL query that asks for tram stops in the specified area

Benefits of Sparqlify The most important benefits of Sparqlify for the LinkedGeoData project are:

- *Simplified ontology modeling and instant feedback through SPARQL interface.* The modelling of vast amounts of data can be altered by adjusting the views, without having to transform billions of triples. For instance, at some point during the evolution of LGD, we decided to follow best practices from GIS science and introduced the distinction between features and geometries in our modelling, which was only a matter of adopting a few Sparqlify-ML view definitions. The effects of this change could be instantly viewed[23]. Although the concept behind this

[23] See `http://linkedgeodata.org/triplify/way5013364` and `http://linkedgeodata.org/geometry/way5013364` for an example of feature-geometry distinction for the Eiffel Tower

is merely the traditional definition of views, which are well known and applied in the relational database world, they have yet to grasp foot in the Semantic Web world.
Another aspect related to views and data modeling is, that there exists multiple RDF vocabularies for describing spatial features, such as WGS84[24], the NeoGeo Geometry[25] or GeoOWL[26]. Although the commitment to certain vocabularies, and efforts to standardize them, reduces the number of published datasets using custom (ad-hoc built) schemas, it still sometimes results in multiple competing schemes, because of the different use cases they cover. RDB-RDF mapping as well as RDF-RDF mapping systems can be used for serving the data in different flavors depending on a client' needs. In our case, we serve point geometries using both the popular WGS84 and the newer NeoGeo Geometry vocabulary.

– *Simplified integration and management of external data sources.* The class/property labels and icons reference tags. Upon changing the mapping between a tag and its corresponding URI, the lable and icons update automatically. ETL based data conversion approaches would have to re-transform data from the external sources based on the modified mappings. In this regard, our approach saves lots of efforts. Also, managing the data in separate tables is often much easier than having to run update operations on a single giant graph. For instance, in RDF it can be very challenging - if not impossible - to pinpoint a specific set of triples, such as for labels, in order to modify them, without affecting any of the remaing ones.
– *Feasibile reification* While maintaining reification statements in RDF is very cumbersome because of its bloat with triples that carry little information, it is actually very easy to achieve with a RDB-RDF mapping system, as shown in Figure 17. Note, that the self join elimination ensures that for every query, whose graph pattern matches that of such a view definition, only yields this view definition's mapping as the result of the rewrite.

```
CREATE VIEW co_occurrence_frequency AS {
  ?a wso:coOccursDirectlyWith ?b .

  ?x owl:annotatedSubject ?a;
     owl:annotatedProperty wso:coOccursDirectlyWith ;
     owl:annotatedObject ?b ;
     wso:frequency ?f .
} WITH
  ?a = uri(concat('http://.../', ?w1_id))
  ?b = uri(concat('http://.../', ?w2_id))
  ?x = uri(concat('http://.../co-oc/',
       ?w1_id, '/', ?w2_id))
  ?f = typedLiteral(?freq, xsd:long)
FROM
  [[SELECT w1_id, w2_id, freq FROM co_n]]
```

Fig. 17: Excerpt from a Sparqlify-ML view definition for the Wortschatz database. The table `co_n` stores information about how often a word with id `w1_id` co-occurs directly with a word identified by `w2_id`. This view definition yields RDF triples that state the fact of the direct co-occurence, and additionally annotations of the triples that state the frequency.

10.2 Crowd sourced mapping with the LGD-Edit

In this section we present the LGD-Edit tool, depicted in Figure 18. It is a web application that enables users to collaboratively edit the different types of tag translations explained in Section 10.1. Thereby users first edit the translations in their *user-branch*, which is isolated from the braches of other users. If they are content with their edits, they can commit their changes to the *master branch*, which then immediatly affects virtual billions of triples. Every user branch has its own virtual SPARQL endpoint[27], which gives instant access to the corresponding RDF data. The data can be explored with the tools provided by the Sparqlify platform, the Pubby Linked Data interface and SNORQL.

Although the tool is still a prototype, it show cases the concept of retaining a separation in the data produced by the OpenStreetMap and Semantic Web communities, yet bridge the gap between them.

The basic idea, to empower the crowd to edit and maintain the specifications of mappings between source data and RDF has already been successfully established for the DBpedia project. There, the *DBpedia Mappings Wiki*[28] enables the Semantic Web community to collaboratively influence the modelling of the ontology and the extraction of RDF data from Wikipedia articles. The difference between Wikipedia and OpenStreetMap lies in the structure of the data: Whereas Wikipedia arcticles are only semi-structured, OpenStreetMap's data consists of tags which are primarily structured data. For the vast majority of the commonly used tags, the community has agreed on (informal) specifications of their use **claim**, which is driven by the need to have clear feature definitions in order to render maps that closely reflect reality.

[24] `http://www.w3.org/2003/01/geo/wgs84_pos`
[25] `http://geovocab.org/geometry`
[26] `http://www.w3.org/2005/Incubator/geo/XGR-geo-20071023/W3C_XGR_Geo_files/geo_2007.owl`
[27] `http://test.linkedgeodata.org/mappings/demo-user/sparql`
[28] `http://mappings.dbpedia.org/`

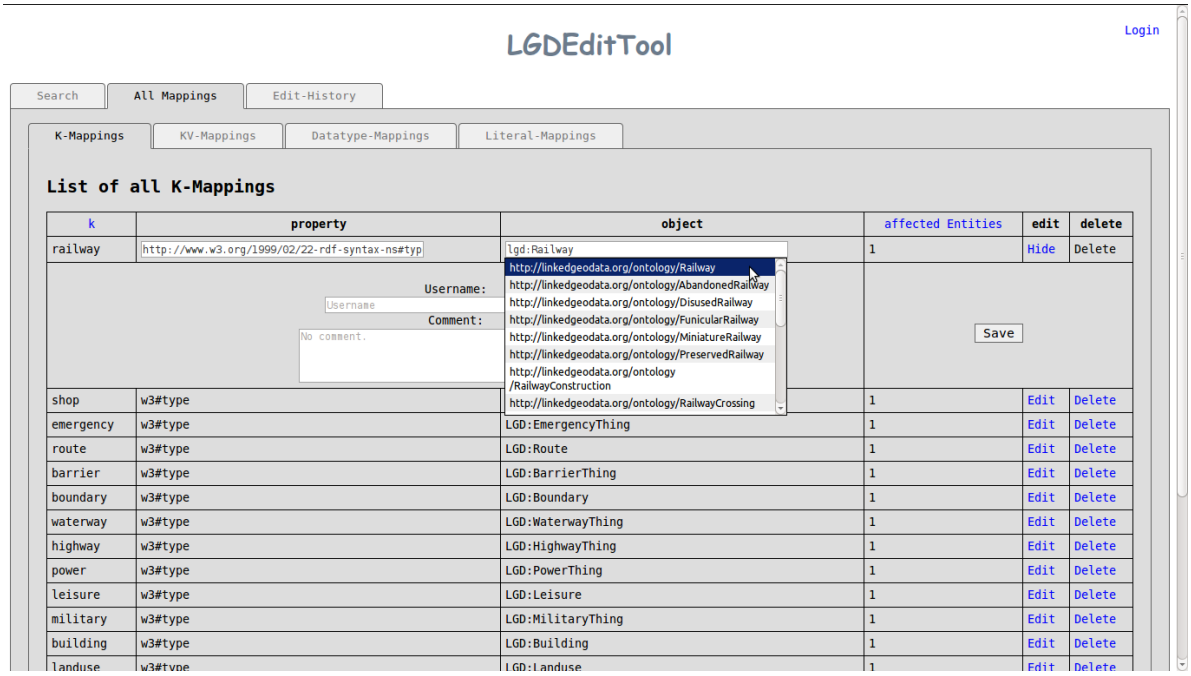

Fig. 18: LinkedGeoData's Tag Mapping Editor.

11 Performance Evaluation

In this section, we discuss the evaluation of Sparqlify using the Berlin Sparql Benchmark (BSBM) [4] and SP2Bench (sp2b) [15]. As a baseline tool *D2R* [3] was chosen, since D2R is the most mature implementation, has a large community used and is also implemented as a standalone application, interacting with a remote database. We selected BSBM because of its widespread use and the provision of both RDF and relational data with corresponding queries in SPARQL and SQL. We used the *Explore* test scenario, consisting of 11 SPARQL queries and their SQL translations. This *Explore* scenario simulates a web shop in which a user browses the product catalog to search for specific products. Datasets were generated with sizes of 1, 25 and 100 million triples. The queries were performed on a single thread with 10 warm-up and 100 measurement runs, using the mapping provided on the BSBM website[29]. In addition to the original BSBM queries, we created a refactored query set. In these queries the resource URI by which BSBM parameterizes a query were factored out of triple patterns and expressed in an equivalent FILTER condition. The factorized queries therefore have the same result sets as the original queries, but reflect a different usage of query patterns.

The sp2b benchmark simulates the analysis of a bibliographic database. In contrast to BSBM, the queries of sp2b are more analytical, with longer runtimes, complex query structures and unbound result set sizes. Due to their complexity, of the 17 queries provided only 9 queries could be executed on both D2R (9 working queries) and Sparqlify (10 working queries). The dataset used in the sp2b run is sized 50,000 and 250,000 triples. The sp2b run was performed with 5 warmup runs and 10 measurement runs.

We used a virtual machine with three AMD Opteron 4184 cores and 4 GB RAM allocated for measurements. Data resided in all cases in a *PostgreSQL* 9.1.3 database[30], with 1 GB of RAM. Additional indices to the BSBM relational schema were added, as described in the BSBM results document[31]. Likewise, we added indices into the sp2b database for all columns generating URIs. Both D2R and Sparqlify were allocated 2 GB of RAM. We utilized D2R version 0.8 and activated in all cases the optimized fast mode.

Figure 19 shows the time totals required for completing the benchmark runs on the different datasets. A first observation is the clear dominance of SQL over D2R and Sparqlify in the smaller datasets. We attribute the slower execution time of SPARQL queries to the overhead imposed by the query translation. With increasing dataset size in the case of Sparqlify this overhead is less significant and consequently the performance gap closes. When relating the execution time

[29] http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/V2/results/store_config_files/d2r-mapping.n3

[30] http://www.postgresql.org/

[31] http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/results/index.html#d2r

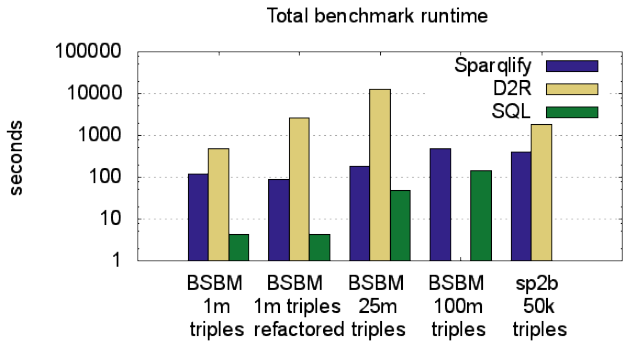

Fig. 19: Total benchmark runtime in seconds on a logarithmic scale.

of the SPARQL-mapper with SQL the superior scaling behavior of Sparqlify can be observed. The factor by which Sparqlify is slower than SQL decreases from 28 (1m) to 3.9 (25m) and 3.3 (100m). For D2R, this factor increases from 111 (1m) to 304 (25m). Consequently the 100 million triple dataset could not be benchmarked, as D2R was running out-of-memory[32]. An additional interesting observation is the increase in execution time for the factorized queries. Due to the normalization step Sparqlify is essentially not affected by such purely syntactical changes of a query, whereas this is the case for D2R.

Using sp2b similar results can be observed. Sparqlify is faster by a factor of 4.5 on the 50k dataset in this case. For the 250k dataset, D2R and Sparqlify both failed on different queries as shown in Figure 21. Examining the benchmark results on a per-query basis, as presented in Figure 20 for BSBM, allows a more detailed discussion of the results. The query execution times of Query 7 and 8 are the main reasons why D2R falls behind Sparqlify in both the 1m and 25m triples scenario. These two queries cause D2R to fetch huge intermediate result sets from the database, which are then processed internally. In the 100m benchmark scenario these result sets do not fit into memory any more and consequently cause the out-of-memory exceptions. Sparqlify, on the other hand, does not suffer form this problem, as it only generates a single SQL query and therefore puts the whole workload on the SQL database. Consequently it achieves a performance comparable to the SQL counterpart.

An interesting observation is related to Query 5, which in its SQL variant is one of the most expensive queries. The corresponding query execution times for Sparqlify and SQL converge with increasing dataset

[32] cf. http://sourceforge.net/mailarchive/message.php?msg_id=28051074

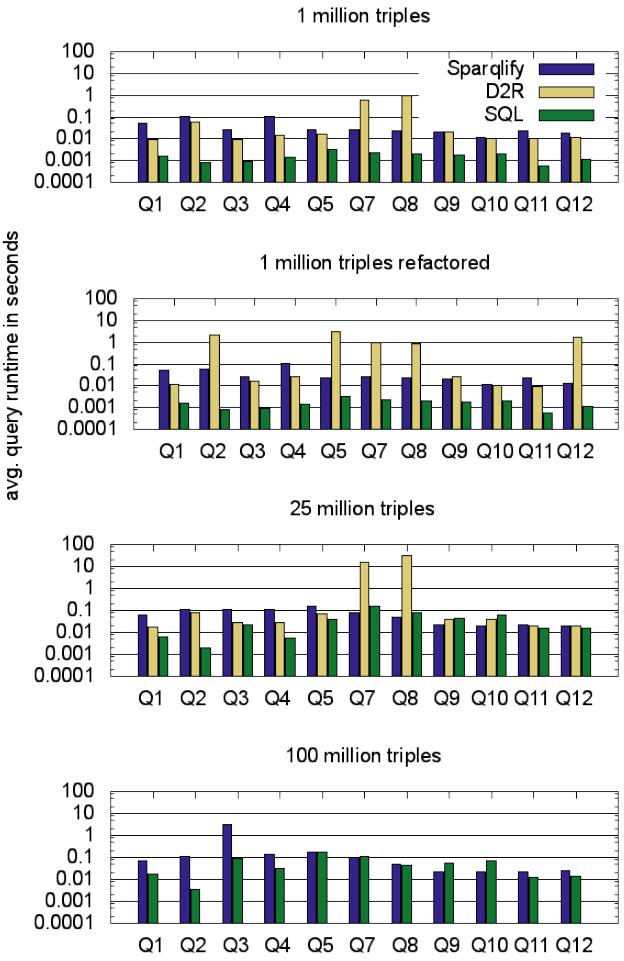

Fig. 20: BSBM benchmark results comparing query runtime of BSBM queries for Sparqlify, raw SQL and D2R with fast mode.

size, which shows that the overhead of the mapping process becomes less significant.

Also noteworthy is, that while most of the queries are faster on D2R, the difference in performance can be attributed to the larger overhead imposed by the Sparqlify approach. This penalty is most visible in the smaller, fast queries of BSBM. A conclusion is that the overhead imposed by the previously described formal framework can have negative effects for smaller datasets and simpler queries, but offers advantages with growing dataset size and query complexity. The results of sp2b, as displayed in Figure 21 fit well into this picture.

The generally higher query executions times of the sp2b queries (minimum 0.05s) reduce the effect of the mapping overhead. In the sp2 benchmark, Sparqlify therefore shows superior performance compared to D2R.

For the 50k sp2b experiment, Sparqlify prevails for all queries with the exception of 3b and 12a. For the

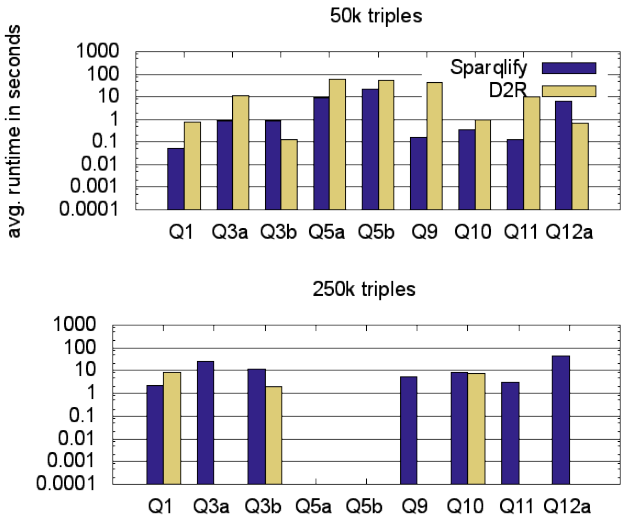

Fig. 21: SP2 benchmark query runtime in seconds on a 50k-triples dataset on a logarithmic scale.

other queries it expensive round-trips between mapper and database lead to performance penalties for D2R. The second benchmark run on a 250k dataset demonstrates the limits of D2R with complex queries. Here 6 of 9 queries fail because of running out of memory, whereas Sparqlify manages to perform 7 out of 9 queries. Sparqlify fails in 2 queries of sp2b because for them PostgreSQL's query execution plan involves storing huge amounts of data in temporary tables.

In summary, D2R outperforms Sparqlify on smaller datasets and less complex queries. However, Sparqlify provides better scalability on large datasets, which is crucial for the mapping large crowdsourced datasets as in the LinkedGeoData use case.

12 Related Work

Related work can be roughly divided into SPARQL-to-SQL conversion as well as crowd-sourcing and the Web of Data.

12.1 SPARQL-to-SQL Conversion

We identified two related areas of research. First, as many native triple stores are based on relational databases, there is considerable research on efficiently storing RDF data in relational schema. Exemplary are both [8] and [6], discussing the translation of a SPARQL into a single SQL query. The translations presented there are, however, targeted towards database backed triple stores and need to be extended and adopted for usage in a database mapping scenario. Also notable is [12], describing SQL structures to represent facets of RDF data in relational databases. Second, the mapping of relational databases into RDF is a way of leveraging existing data into the Semantic Web. We can differentiate between tools that either expose their data as RDF, Linked Data or expose a SPARQL endpoint for interactive querying of the data. An example for RDF and Linked Data exposition is *Triplify* [1]. Exposing data via a SPARQL endpoints either requires loading transformed data into a SPARQL-enabled triple store or rewriting SPARQL queries into SQL. The answering of SPARQL queries over relational data is the goal of several concepts and implementations. *D2R Server* [3] is a standalone web application, answering SPARQL queries by querying the mapped database. D2R mixes in-database and out-of-database operations. Operators of an incoming SPARQL queries like joins and some filters are performed in the mapped database directly. Other operators are then later executed on the intermediate results directly by D2R. OpenLink's *Virtuoso RDF Views* [18] allows the mapping of relational data into RDF. RDF Views are integrated into the Virtuoso query execution engine, consequently allowing SPARQL query over native RDF and relational data. A further SPARQL-to-SQL tool is *Ultrawrap* [16], which integrates closely with the database and utilizes views for answering SPARQL queries over relational data. A list of RDB2RDF implementations is maintained by the corresponding W3C working group at `http://w3.org/TR/rdb2rdf-implementations/`.

12.2 Crowd-sourcing and the Web of Data

Apart from LinkedGeoData, there are several other projects, which convert data from a crowd sourced community project to RDF and Linked Data. One of the most prominent efforts is DBpedia [10, 14], which converts articles in Wikipedia to RDF. In this case, the challenge is to understand Wiki syntax and convert it to a common ontological schema. In contrast to LinkedGeoData, the focus of the DBpedia project is more on cleaning the obtained data and designing Wiki system specific extractors. Other projects, which also extract information from Wikis are Wiktionary RDF[33] and YAGO [9]. Another approach which is closer to LinkedGeoData is the RDF version of Musicbrainz[34] and Jamendo[35]. Both share similar characteristics with OpenStreetMap, i.e. a relational database backend with support for stor-

[33] `http://wiki.dbpedia.org/Wiktionary`
[34] `http://dbtune.org/musicbrainz/`
[35] `http://dbtune.org/jamendo/`

ing tags, and are currently backed by D2R as mapping tool. In general, we believe this setup is common in community based crowdsourced data collection efforts, since it is very flexible and easy to understand for users.

13 Conclusion and Future Work

In this article we presented the *Sparqlify* approach, which comprises of:

- The simple, yet powerful mapping language *Sparqlify-ML* that reuses familiar elements of SPARQL and SQL in order to lower the learning curve and ease the manual writing of view definitions.
- A comprehensive formalization of the RDB-RDF rewriting process based the definition of RDF views of a relational database and bindings, which bridge between the relational and SPARQL algebra.
- The SPARQL-SQL rewriter engine, which generates optimized SQL queries using unsatisfiablity checks and self-join elimination.

Sparqlify has been successfully deployed in the LinkedGeoData project on the OpenStreetMap database[36], where it gives access to more than 20 billion RDF triples backed by about 3 billion relational rows. Our evaluation in the LinkedGeoData testbed as well as with the synthetic BSBM and SP2 benchmarks show, that Sparqlify delivers comparable performance with small datasets but substantially increased scalability when compared to the state-of-the-art. In the future, the query generation overhead can be further substantially reduced by enabling prepared SPARQL queries, where a SPARQL query template is already precompiled into the corresponding SQL query template and subsequently reoccurring queries using the template do not have to be translated anymore.

This work is the first step in a larger research agenda: In the future, we hope to be able to remove even more barriers between the relational and RDF worlds. Currently, our mapping approach one-directional and read-only: queries are translated from SPARQL to SQL and query results from SQL to SPARQL. In the future, we plan to make the RDB-RDF mapping writable by enabling SPARQL updates, which are translated into updates of the underlying relational database. An interesting research direction, for which our formalization could be exploited, are bi-directional and hybrid mapping allowing an RDBMS to access RDF data and a possible incorporation of relational and RDF sources in a single mapping. A further research question is how inferencing and reasoning features can be integrated into the RDB-RDF mapping and rewriting. Another issue in this regard is how the SPARQL 1.1 features[37], such as property paths of arbitrary length, can be efficiently supported in the SPARQL-SQL rewriting. Furthermore, although we believe that Sparqlify-ML is easier to use while offering the very similar functionality as the RDF based mapping language R2RML[38], the latter has been standardized very recently and should therefore be supported by the Sparqlify system. Yet, we believe that Sparqlify-ML is in most cases easier to learn and use than R2RML, and therefore see future efforts on improving Sparqlify-ML as complementary.

[36] Using the "simple schema" provided and supported by the respective OpenStreetMap tools.

[37] `http://w3.org/TR/sparql11-query/`

[38] `http://w3.org/TR/r2rml/`

References

1. Auer, S., Dietzold, S., Lehmann, J., Hellmann, S., Aumueller, D.: Triplify: Light-weight linked data publication from relational databases. In: J. Quemada, G. León, Y.S. Maarek, W. Nejdl (eds.) Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009, pp. 621–630. ACM (2009). DOI doi:10.1145/1526709.1526793. URL `http://doi.acm.org/10.1145/1526709.1526793`
2. Auer, S., Lehmann, J., Hellmann, S.: LinkedGeoData - adding a spatial dimension to the web of data. In: Proc. of 8th International Semantic Web Conference (ISWC) (2009). DOI doi:10.1007/978-3-642-04930-9_46. URL `http://www.informatik.uni-leipzig.de/~auer/publication/linkedgeodata.pdf`
3. Bizer, C., Cyganiak, R.: D2r server – publishing relational databases on the semantic web. Poster at the 5th Int. Semantic Web Conf. (ISWC2006) (2006). URL `http://www4.wiwiss.fu-berlin.de/bizer/pub/Bizer-Cyganiak-D2R-Server-ISWC2006.pdf`
4. Bizer, C., Schultz, A.: The berlin SPARQL benchmark. Int. J. Semantic Web Inf. Syst **5**(2), 1–24 (2009). URL `http://www.igi-global.com/articles/details.asp?ID=35035`
5. Ceri, S., Gottlob, G.: Translating SQL into relational algebra: Optimization, semantics and equivalence of SQL queries. IEEE Transactions on Software Engineering **11**(4), 324–345 (1985)
6. Chebotko, A., Lu, S., Fotouhi, F.: Semantics preserving sparql-to-sql translation. Data and Knowledge Engineering **68**(10), 973 – 1000 (2009). DOI 10.1016/j.datak.2009.04.001. URL `http://www.sciencedirect.com/science/article/pii/S0169023X09000469`
7. Cyganiak, R.: A relational algebra for SPARQL. Tech. rep., Digital Media Systems Laboratory, HP Laboratories Bristol (2005)
8. Elliott, B., Cheng, E., Thomas-Ogbuji, C., Ozsoyoglu, Z.M.: A complete translation from sparql into efficient sql. In: Int. Database Engineering & Applications Symp., IDEAS '09, pp. 31–42. ACM (2009). DOI 10.1145/1620432.1620437. URL `http://doi.acm.org/10.1145/1620432.1620437`
9. Hoffart, J., Suchanek, F.M., Berberich, K., Weikum, G.: Yago2: A spatially and temporally enhanced

knowledge base from wikipedia. Artificial Intelligence (2012). DOI 10.1016/j.artint.2012.06.001. URL http://www.sciencedirect.com/science/article/pii/S0004370212000719

10. Lehmann, J., Bizer, C., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S.: DBpedia - a crystallization point for the web of data. Journal of Web Semantics **7**(3), 154–165 (2009). DOI doi:10.1016/j.websem.2009.07.002. URL http://jens-lehmann.org/files/2009/dbpedia_jws.pdf
11. Le, W., Duan, S., Kementsietsidis, A., Li, F., Wang, M.: Rewriting queries on sparql views. In: S. Srinivasan, K. Ramamritham, A. Kumar, M.P. Ravindra, E. Bertino, R. Kumar (eds.) WWW, pp. 655–664. ACM (2011). URL http://dblp.uni-trier.de/db/conf/www/www2011.html#LeDKLW11
12. Lu, J., Cao, F., Ma, L., Yu, Y., Pan, Y.: Semantic web, ontologies and databases. chap. An Effective SPARQL Support over Relational Databases, pp. 57–76. Springer-Verlag, Berlin, Heidelberg (2008). DOI 10.1007/978-3-540-70960-2_4. URL http://dx.doi.org/10.1007/978-3-540-70960-2_4
13. Manola, F., Miller, E.: RDF primer. World Wide Web Consortium, Recommendation REC-rdf-primer-20040210 (2004)
14. Morsey, M., Lehmann, J., Auer, S., Stadler, C., Hellmann, S.: Dbpedia and the live extraction of structured data from wikipedia. Program: electronic library and information systems **46**, 27 (2012). URL http://svn.aksw.org/papers/2011/DBpedia_Live/public.pdf
15. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: Sp2bench: A sparql performance benchmark. CoRR **abs/0806.4627** (2008)
16. Sequeda, J.F., Miranker, D.P.: Ultrawrap: SPARQL Execution on Relational Data. Poster at the 10th Int. Semantic Web Conf. (ISWC2011) (2011). URL https://files.ifi.uzh.ch/ddis/iswc_archive/iswc/ab/2011pre/iswc2011.semanticweb.org/fileadmin/iswc/Papers/PostersDemos/iswc11pd_submission_94.pdf
17. Stadler, C., Lehmann, J., Höffner, K., Auer, S.: Linkedgeodata: A core for a web of spatial open data. Semantic Web Journal (2011)
18. Mapping relational data to rdf with virtuoso's rdf views. http://virtuoso.openlinksw.com/whitepapers/relational%20rdf%20views%20mapping.html