© 20XX by the authors; licensee RonPub, Lübeck, Germany. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/4.0/).

Open Access

Open Journal of Semantic Web (OJSW)
Volume X, Issue X, 20XX

http://www.ronpub.com/ojsw
ISSN 2199-336X

OnGIS: Semantic query broker for heterogeneous geospatial data sources

Marek Šmíd [A], Petr Křemen [A]

[A] Faculty of Electrical Engineering, Czech Technical University in Prague, Technická 2, 166 27 Prague, Czech Republic, {marek.smid, petr.kremen}@fel.cvut.cz

Abstract

Querying geospatial data from multiple heterogeneous sources backed by different management technologies poses an interesting problem both in the data integration and the subsequent result interpretation. This paper proposes broker techniques for answering a user's complex spatial query: find relevant data sources (from a catalogue of many) capable of answering the query, eventually split the query and find relevant sources for the query parts, when no single source suffices. For the purpose, we describe each source with a set of prototypical queries that are algorithmically arranged into a lattice, which makes searching efficient. The proposed algorithms leverage GeoSPARQL query containment enhanced with OWL 2 QL semantics. A prototype is implemented in a system called OnGIS.

Type of Paper and Keywords

Regular research paper: *geospatial semantics, OWL 2 QL, data integration, query containment, query broker*

1 Introduction

OnGIS is a semantic geospatial query broker. Some parts of it (simple semantic data integration, two options of user interface) has been previously developed as prototypes, see [14, 15].

We set ourselves two goals when designing a semantic query broker: to choose a universal language, which would be used for describing what data different sources contain (i.e. a language appropriate for data integration), and to find a way of describing what operations the sources can do with their data.

We solved both the tasks with describing the sources with sets of semantic prototypical queries they can answer. Prototypical queries fully capture relevant capabilities of a data source, both in terms of data and operations it provides. It is necessary, since heterogeneous sources and services are considered (therefore no single specific technology for describing capabilities can be used), e.g. (a) traditional GIS web services (OGC – Open Geospatial Consortium[1], a large consortium, is setting many standards, e.g. WFS – Web Feature Service [10], a standard for publishing vector spatial data, WCS – Web Coverage Service [12], a standard for publishing source raster data, ESRI ArcGIS services, etc.), (b) plain relational databases (PostgreSQL+PostGIS, Oracle Spatial, etc.), and (c) Linked Data sources available through SPARQL [21] endpoints (DBpedia, LinkedGeoData, etc.). Each can serve different data, and have different capabilities.

Linked Data[2] is a promising method for publishing and integrating different kinds of data including spatial data. For example, OpenStreetMap data are already

[1] `http://www.opengeospatial.org/`, cit. 21.6.2016.
[2] `http://linkeddata.org/`, cit. 21.6.2016.

published this way as LinkedGeoData[3]. We will borrow some of Linked Data techniques. First of all, as the language used for describing the prototypical queries, GeoSPARQL (see Section 3.1) is used. It is a recent, detailed, spatial-query-enabling extension of SPARQL, a query language for data in RDF – Resource Description Framework [23], a standard model for data interchange (which is also the primary format of Linked Data).

The idea of how to choose a data source for answering a query is that the system takes all prototypical queries of all the sources, form a special data structure with them, and when a user asks a query, it compares the user's query with the data structure and decides, which data source(s) to use to answer the user's query. For comparing queries together we use a method called query containment, for which we utilize the OWL 2 QL language, one of the OWL languages used for ontologies, which goes beyond the expressive power of RDF alone.

In comparison with our previous work, and the work of others, OnGIS supports semantic discovery of geospatial sources capable of answering parts of a query.

In Section 2 we will briefly explore existing methods and technologies, Section 3 gives background on GeoSPARQL and OWL 2 QL, Section 4 contains all necessary parts of query containment we have designed (how to compute it, including the GeoSPARQL semantics), Section 5 describes how to construct and search a lattice of queries, Section 6 gives an overall example, and finally Section 7 concludes the text.

2 Related Work

There are some techniques for defining and searching GIS catalogues. For example the Catalogue Service (CSW), a standard by OGC, is an interface to discover, browse, and query metadata about GIS data and services. It uses Dublin Core[4] vocabulary to describe web resources. It can be searched by metadata – keywords, author, date, etc. However, it does not allow for more complex queries or semantic search; for spatial querying only bounding box is supported – the language for describing the source's capabilities is very limited.

There is also some work on using semantic technologies for spatial data. In [24], an ontology-based information system is implemented, focusing on ontology-based spatio-thematic query answering for city maps. It bases on Description Logics reasoner RacerPro [6], implementing more expressive logic (compared to what we use) $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D}^-)$. The system implements its own custom storage, which directly includes the inference algorithms and the query evaluation engine. A custom query language, SuQL, is used. However, it does not solve the problem of integrating multiple data sources.

[3] `http://linkedgeodata.org/`, cit. 21.6.2016.

[4] `http://dublincore.org/`, cit. 21.6.2016.

The authors in [3] use Parliament triple store, supporting geospatial indexes, for storing spatial data and for making complex spatial queries via GeoSPARQL over them. However they use a precomputed data set and do not directly support data integration.

The system in [28] links an RDF ontology to databases and WFS. It uses custom rules and algorithms for query rewriting, but it does not provide the standard OWL semantics. However, it supports query answering from multiple data sources, specifically WFS servers for spatial data and databases (via the D2R interface) for attributes.

The spatial decision support system in [26] integrates various data sources (OGC standards WMS, WFS, WCS, WPS) and links them with ontologies. It also uses catalogue services via ontologies and automatic web service discovery. However, it focuses more on geospatial analysis and ontology alignment than spatial search.

Similar goal as OnGIS have the authors of [9], where they propose interesting system based also on semantic technologies. But instead of open-world OWL semantics, they use rules (specifically SWRL rules), both for integrating sources and for answering queries. Also the whole problem of data integration is summarized there.

Buster [20, 19] is a complex system dealing with terminological, spatial, and temporal query answering. It provides a common interface to heterogeneous information sources in terms of an intelligent information broker. It represents its terminology using the language OIL and the description logic $\mathcal{SHIQ}$ (it is utilizing the FaCT reasoner), and uses Dublin Core as the vocabulary for modeling metadata. It solves some of OnGIS goals in a similar fashion, but it does not support complex queries in terms of e.g. spatial joins, and it does not support participating multiple sources on one user's query.

The system Karma presented in [27] uses its own base linking ontology for integrating spatial data sources. The linking ontology seems rather limited, as opposed to standard Simple Feature and GML ontologies accompanying GeoSPARQL. It also performs linking of features in two data sources by their spatial similarity. However, it uses only limited RDF expressive power and does not support complex spatial queries.

Comprehensive overview of related work in the area of ontology-driven GIS integration is in [4].

As query containment is an important part of our query-broking solution (see Section 4), query containment capabilities of several systems have been examined. FaCT++ [17] is one of the reasoners with the support for query containment, but it does not support custom datatypes. Unfortunately, description of how the query containment works could not be found.

Pellet[5] is another reasoner with query containment support. But it has a problem with data properties, since all variables in its query containment module are modeled as individuals: when there is a data property with a variable, there is a problem with illegal punning in OWL. Pellet also supports the query language SPARQL-DL [13], a SPARQL subset with OWL-based semantics.

The -ontop- system[6], specifically its SPARQL query engine Quest, has some support for query containment, however it seems it is used only for removing redundant queries during query rewriting; not much information is available.

SPIN[7], which stands for SPARQL Inferencing Notation, is a SPARQL-based rule and constraint language able to represent arbitrary SPARQL query in RDF. But the SPIN RDF representation of a query seemed unsuitable for deciding query containment with OWL semantics to us. For example, there is a problem with an OWL-illegal punning: SPIN uses the same property[8] for linking both to a resource (IRI), and a literal, which means the property would be both data property and object property, which is illegal in OWL.

3 BACKGROUND

Choosing GeoSPARQL as the language for describing spatial queries is a logical choice, as it is the most detailed and up-to-date standard for describing spatial queries over semantic data (specifically RDF). It is based on RDFS reasoning. However, for deciding query containment of GeoSPARQL queries, we use technique requiring logical negation, which is not contained in RDFS. Therefore, we chose OWL 2 QL, an ontology language suitable for its trade-off between expressive power (it adds a few features to RDFS, including limited negation, which suffices to our purposes), and computational properties (it is tractable, i.e. evaluable using a relational database). OWL 2 QL is therefore suitable also as the language for querying data from the respective sources.

3.1 GeoSPARQL

GeoSPARQL is a relatively recent standard published by OGC [11]. It extends SPARQL query language for RDF data, adding support for spatial data and spatial operations.

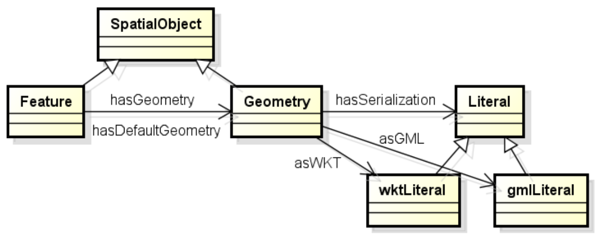


Figure 1: GeoSPARQL basic classes and properties.

Basic GeoSPARQL concepts are in Fig. 1: three basic classes: `SpatialObject`, which has two disjoint sub-classes `Feature` and `Geometry`. The object property `hasDefaultGeometry` is a sub-property of `hasGeometry`, which links the two sub-classes.

`Geometry` instances can have various data properties. The most important is the data itself, given in a form of serialization. All serialization data properties are sub-properties of `hasSerialization`, and the predefined ones are `asWKT` for WKT strings, having the range of custom datatype `wktLiteral`, and `asGML` for GML data, having the range of custom datatype `gmlLiteral`. Other datatype properties are `dimension` (topological dimension), `coordinateDimension` (dimension of direct positions), `spatialDimension` (dimension of the spatial portion of the direct positions), `isEmpty` (has no points), and `isSimple` (contains no self-intersections except of its boundary).

Then it defines object properties for topological relations, there are three definition families of the relations, each containing eight relations. Every such object property has its domain and range equal to `SpatialObject`.

- *Simple Feature* family contains `sfEquals`, `sfDisjoint`, `sfIntersects`, `sfTouches`, `sfWithin`, `sfContains`, `sfOverlaps`, `sfCrosses`.
- *Egenhofer* family contains `ehEquals`, `ehDisjoint`, `ehMeet`, `ehOverlap`, `ehCovers`, `ehCoveredBy`, `ehInside`, `ehContains`.
- *RCC8 – Region Connection Calculus*[9] family contains `rcc8eq` (equals), `rcc8dc` (disconnected), `rcc8ec` (externally connected), `rcc8po` (partially overlapping), `rcc8tppi` (tangential proper part inverse), `rcc8tpp` (tangential proper part), `rcc8ntpp` (non-tangential proper part), `rcc8ntppi` (non-tangential proper part inverse).

[5] `http://clarkparsia.com/pellet/`, cit. 20.5.2014.
[6] `http://ontop.inf.unibz.it/`, cit. 4.6.2014.
[7] `http://spinrdf.org/`, cit. 10.5.2014.
[8] `http://spinrdf.org/sp#object`

[9] Simple description is at `http://en.wikipedia.org/wiki/Region_connection_calculus`, cit. 22.5.2014.

All the three families divide all possible spatial relations between two objects into eight basic topological relations, but not exactly the same way. The precise meaning of each topological relation can be described by the DE-9IM model, which uses 3×3 matrices; for details see [5].

There is also a set of GeoSPARQL functions that can be used in the filter section as defined in SPARQL specification. These functions include alternates of all topological relation properties, to be applied as functions on geometry literals.

But there are also some more complex functions for comparing and manipulating geometries, e.g. `distance` for obtaining distance between two geometry literals (only this one is currently supported in our prototype), `union`, `intersection`, `difference`, and some others. GeoSPARQL also contains some RIF (Rule Interchange Format) [22] rules, but we will not consider them as well.

GeoSPARQL vocabulary and definitions are contained in an ontology provided by OGC[10]. In the rest of this text, we will refer to the main GeoSPARQL namespace[11] with the prefix `geo:`.

3.2 OWL 2 QL

OWL 2 QL [25] is a profile of the Web Ontology Language (OWL). The key feature is its tractability (along with other OWL 2 profiles) traded for expressiveness, which is lower compared e.g. to OWL 2 DL. The tractability allows reformulation of description logic queries into SQL and thus RDBMSs (relational database management systems) can be used as OWL 2 QL storage.

OWL 2 QL is based on $DL{-}Lite^{\mathcal{H}}_{core}$, a member of the $DL{-}Lite$ family of description logics [2] defined in [1]. $DL{-}Lite^{\mathcal{H}}_{core}$ constructs for defining concepts and roles in description logics syntax are:

$$B ::= A \,|\, \exists R, \;\; C ::= B \,|\, \neg B, \;\; R ::= P \,|\, P^{-},$$

where A denotes a named concept, B a basic concept, and C a general concept. Symbol P denotes a named role, and R a complex role.

The semantics is defined by an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a nonempty interpretation domain and $\cdot^{\mathcal{I}}$ is an interpretation function, that assigns to each individual an element of $\Delta^{\mathcal{I}}$, to each named concept a subset of $\Delta^{\mathcal{I}}$, and to each named role a binary relation over $\Delta^{\mathcal{I}}$. Semantics of the used constructs are defined in Table 1.

[10] Available at `http://schemas.opengis.net/geosparql/`, cit. 16.4.2014, alongside with the imported ontologies for Simple Feature and GML geometries.

[11] `http://www.opengis.net/ont/geosparql#`

Table 1: Constructs used in $DL{-}Lite$ and their semantics

Syntax	Semantics	Comment
A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$	named concept
P	$P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$	named role
P^{-}	$(P^{-})^{\mathcal{I}} =$ $= \{(b,a) \vert (a,b) \in P^{\mathcal{I}}\}$	inverse of a role
$\exists R$	$(\exists R)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \vert \exists b :$ $: (a,b) \in R^{\mathcal{I}}\}$	existential quantification
$\neg B$	$(\neg B)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus B^{\mathcal{I}}$	negation of a basic concept

Table 2: ABox axioms used in $DL{-}Lite$ and their semantics

Syntax	Semantics	Comment
a, b	$a^{\mathcal{I}}, b^{\mathcal{I}} \in \Delta^{\mathcal{I}}$	individuals
$A(a)$	$a^{\mathcal{I}} \in A^{\mathcal{I}}$	concept assertion
$P(a,b)$	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \in P^{\mathcal{I}}$	property assertion

A TBox (a set of all ontology terminological axioms — subsumptions of concepts, domains, etc.) can be defined by inclusion axioms of the form: $B \sqsubseteq C$, and $R_1 \sqsubseteq R_2$, interpreted by $\mathcal{I}$ as $B^{\mathcal{I}} \subseteq C^{\mathcal{I}}$, resp. $R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$.

An ABox consists of the following assertion axioms: $A(a)$, and $P(a,b)$, where a, b are individuals. The semantics of the axioms as interpreted by $\mathcal{I}$ is in Table 2.

OWL 2 QL extends $DL{-}Lite$ with various features not affecting its tractability, e.g. data roles.

In OWL terminology, concepts are called classes, and roles are called properties. We will use OWL terminology in the rest of this text.

4 Query Containment with GeoSPARQL

The essential part of matching a query against a set of queries is the problem called query containment, which decides subsumption relation of two queries. One query is subsumed by another, $q_1 \sqsubseteq q_2$, whenever each result set of q_1 for data D is a subset of the result set of q_2 for the same data D.

We formulate the queries in a subset of SPARQL language, with the extension of some of the GeoSPARQL vocabulary and its semantics, and the OWL 2 QL semantics. From the SPARQL language, only *SELECT* queries are supported. Graph patterns and filters are supported, however optional patterns, ordering, grouping, offsets and limits are not.

From the GeoSPARQL query language, besides basic classes, properties, and serializations, topological relations and some functions are supported. Specifically, the supported features are:

- classes `SpatialObject`, `Feature` and `Geometry`,
- object properties `hasGeometry` and `hasDefaultGeometry`,
- data properties `hasSerialization`, `asWKT`, and `asGML` (and parsing its literals in WKT and GML),
- all the three topological relation families (Simple Feature, Egenhofer, and RCC8),
- and the `distance` function (for details of all the above, see Section 3.1).

In Section 4.1, hierarchy on topological relations is defined, then Section 4.2 defines general query containment algorithm for OWL 2 QL, and Section 4.3 gives reasoning extension for GeoSPARQL.

4.1 Expanding GeoSPARQL ontology

One part of supporting semantics of GeoSPARQL is to understand the relations between the topological relations. GeoSPARQL ontology contains only a list of all the topological relations, without any hierarchy. Therefore, we added a hierarchy comparing topological relations between different relation families, in order to support deciding query containment of queries using them. Naturally, topological relation inside a family are mutually exclusive, and thus does not form a non-trivial hierarchy. For example, both `rcc8tpp` and `rcc8ntpp` (tangential and non-tangential proper part from the RCC8 family) are sub-properties of `sfWithin`.

The complete hierarchy of relations was determined by their DE-9IM definitions, and its visualisation in Protégé[12] is in Fig. 2.

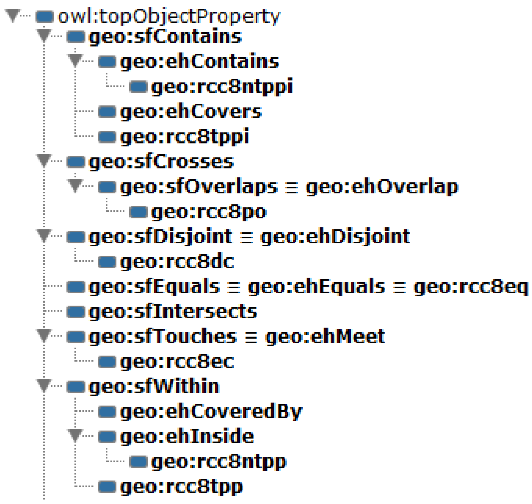


Figure 2: Extending GeoSPARQL with the hierarchy of topology object relations.

4.2 Query containment basics

First, let us define a query as a tuple of output variables, class restrictions, object and data property restrictions, and filters, formally

$$q = (V_o, R_c, R_{op}, R_{dp}, R_f),$$

where

- V_o is a set of output variables of the query; a variable is in the rest of the text prefixed with the question mark,
- R_c is a set of class restriction on variables, $C(?x)$, where C is a class name,
- R_{op} is a set of object property restrictions in the form of either $OP(?x, ?y)$, $OP(?x, i)$, or $OP(i, ?y)$, where OP is an object property name, and i is an individual,
- R_{dp} is a set of data property restrictions in the form of either $DP(?x, ?y)$, $DP(?x, d)$, or $DP(i, ?y)$ where DP is a data property name, and d is a literal,
- R_f is a set of filters, restricting the result by functions. A filter is a predicate (a function returning a boolean value), which must be satisfied for the returned results. It has the form of $f(\underline{x})$, where f is a boolean function, and $\underline{x}$ is a tuple having the same arity as f and the proper types. Elements of $\underline{x}$ can be variables, individuals, literals, and other function calls.

The elements of R_c, R_{op} and R_{dp} are also called *triples* of the query q.

The main idea, how to decide query containment, is based on [8] and [7]: take the two compared queries, q_1 and q_2, and a background ontology $\mathcal{O}$ (TBox and ABox of valid axioms, on which background the query containment is to be decided) and transform the queries into two ABoxes, which, using a series of satisfiability checks, lead to deciding whether $\mathcal{O} \models q_1 \sqsubseteq q_2$.

[12] An open-source ontology editor, `http://protege.stanford.edu/`, cit. 22.5.2014

First, we define a canonical ABox of a query (it is basically just substituting variables with individuals, one individual per variable):

$$\begin{aligned}
\text{Can}(q) &= \text{Can}(R_c) \cup \text{Can}(R_{op}) \cup \text{Can}(R_{dp}) \\
\text{Can}(R_c) &= \{C(i_x) : C(?x) \in R_c\} \\
\text{Can}(R_{op}) &= \{OP(i_x, b) : OP(?x, b) \in R_{op}\} \\
&\cup \{OP(a, i_y) : OP(a, ?y) \in R_{op}\} \\
&\cup \{OP(i_x, i_y) : OP(?x, ?y) \in R_{op}\} \\
\text{Can}(R_{dp}) &= \{DP(i_x, d) : DP(?x, d) \in R_{dp}\} \\
&\cup \{DP(a, d_y) : DP(a, ?y) \in R_{dp}\} \\
&\cup \{DP(i_x, d_y) : DP(?x, ?y) \in R_{dp}\}
\end{aligned}$$

where a subscripted i is a fresh individual, C is a class, OP is an object property, DP is a data property, a subscripted d is a fresh (artificial) literal, and a letter prefixed with the question mark is a query variable.

Let us denote $A_1 = \text{Can}(q_1)$, $A_2 = \text{Can}(q_2)$, I_1 all individuals in A_1, I_{V1} all individuals representing variables in A_1, D_1 all literals in A_1, D_{V1} all literals representing variables in A_1, similarly I_2, I_{V2}, D_2, and D_{V2} for A_2, and $\mathcal{O}$ a background ontology.

A *completed ABox of a property* is an extended ABox built on top of canonical ABoxes:

$$\begin{aligned}
&\text{Com}(OP(i_x, i_y), q_1, q_2) = \\
&= \{(i'_x, i_y) : i'_x \in I_1 \setminus I_{V1}\} \text{ when } i_x \in I_{V2} \\
&\cup \{(i_x, i'_y) : i'_y \in I_1 \setminus I_{V1}\} \text{ when } i_y \in I_{V2} \\
&\text{Com}(DP(i_x, d_y), q_1, q_2) = \\
&= \{(i'_x, d_y) : i'_x \in I_1 \setminus I_{V1}\} \text{ when } i_x \in I_{V2} \\
&\cup \{(i_x, d'_y) : d'_y \in D_1 \setminus D_{V1}\} \text{ when } d_y \in D_{V2}
\end{aligned}$$

Using those, we can define a *testing ontology* $\bar{\mathcal{O}}(a)$ as a function of axioms from A_2:

$$\begin{aligned}
&\bar{\mathcal{O}}(C(i_x)) = \\
&\quad = (\mathcal{O} \cup A_2) \setminus \{C(i_x)\} \cup A_1 \cup \{NC(i_x)\} \\
&\bar{\mathcal{O}}(OP(i_x, i_y)) = \\
&\quad = (\mathcal{O} \cup A_2) \setminus \{OP(i_x, i_y)\} \cup A_1 \\
&\quad \cup \{NOP(i_x, i_y)\} \\
&\quad \cup \{NOP(i'_x, i'_y) : (i'_x, i'_y) \in \text{Com}(OP(i_x, i_y))\} \\
&\bar{\mathcal{O}}(DP(i_x, d_y)) = \\
&\quad = (\mathcal{O} \cup A_2) \setminus \{DP(i_x, d_y)\} \cup A_1 \\
&\quad \cup \{NDP(i_x, d_y)\} \\
&\quad \cup \{NDP(i'_x, d'_y) : (i'_x, d'_y) \in \text{Com}(DP(i_x, d_y))\},
\end{aligned}$$

where NC is a fresh class for each C with the restriction $NC \sqsubseteq \neg C$ added to $\bar{\mathcal{O}}(a)$, and similarly for NOP for each OP and NDP for each DP.

Then if there exists an assertion $a \in A_2$ such that $\bar{\mathcal{O}}(a)$ is consistent, or filters are not contained (see below), then $q_1 \not\sqsubseteq q_2$, otherwise $q_1 \sqsubseteq q_2$.

The proof of the correctness can be based on proofs in [8] and [7], as our steps are based on those articles with suitable modifications and simplifications for OWL 2 QL semantics. Detailed proofs are out of scope of this text.

Intuitive proof: in order to $q_1 \sqsubseteq q_2$ be valid, q_1 has to restrict results more than q_2. This is tested by taking one restriction in q_2 (as an axiom in A_2) at a time, negating it, and putting it together with the background ontology, A_1, and the rest of A_2; and in case of a property, also some additional axioms. This altogether is tested for consistency; if it is consistent, it means that results are less restricted by q_1 than by q_2 (q_1 still has some results, even if it is restricted with the negation of a q_2 restriction, meaning skipping at least the results originally given by q_2), and therefore $q_1 \not\sqsubseteq q_2$. The additional axioms mentioned above are given by $\text{Com}(OP)$ and $\text{Com}(DP)$; they are necessary because a variable, in A_2 represented by an individual/literal, can be substituted by any non-variable individual/literal. For consistency checks in the query containment decisions it is necessary to substitute only non-variable individuals/literals in A_1.

To deal with filters, different approach has to be used. They have multiple arity and they have different semantics (they do not posses open world assumption), thus using their reification, negation, and consistency checks does not solve their containment.

A simple scheme is used for deciding query containment with filters:

$$(\exists f_2 \in R_{f2} : \nexists f_1 \in R_{f1} : f_1 \sqsubseteq f_2) \Rightarrow q_1 \not\sqsubseteq q_2.$$

Intuitively the filter containment relation $\sqsubseteq$ expresses whether one filter is more restrictive than the other. It has to be defined according to the specific filter function definition.

Another problem in query containment is variable mapping. Using the simplest attitude, variables are converted to individuals and literals, assuming they are uniquely identified by their names, and the query containment algorithm decides the query subsumption. It works, if the variables are named consistently between the compared queries. However, this might not be the case in the real world – different systems may generate the prototypical queries describing its source capabilities, and they may name the variables differently. One option is to check all possible ways how to rename the variables of one query to the variables of the other while deciding query containment; when at least one combination results in the positive answer, one query is contained within the other. In OnGIS, we implemented a simple algorithm to reduce the number of combinations to check the containment for, but we will not discuss it in this article.

4.3 Adding support for GeoSPARQL

When deciding query containment with queries only on symbolic level with individuals, topological relations are covered by extending them with a hierarchy, as in Section 4.1. But when there are geospatial literals involved, it gets more complicated.

First, we define the set of all topological relation restrictions of a geometry individual as

$$\begin{aligned} &Rel(i, q_2) = \\ &= \{-TR(OP) : OP(i_x, i_y) \in R_{op2} \\ &\quad \wedge (hG(i_x, i) \vee i_x = i)\} \\ &\cup \{TR(OP) : OP(i_x, i_y) \in R_{op2} \\ &\quad \wedge (hG(i_y, i) \vee i_y = i)\}, \end{aligned}$$

where $hG \sqsubseteq$ `geo:hasGeometry`, R_{op2} is R_{op} in q_2, and $TR(OP)$ (topological relation restriction values for all topological relations) is defined in Table 3. Note that in a topological relation both a geometry individual and a feature individual can play roles (thus the logical *or* in the definition).

The numerical values of $TR(OP)$ in Table 3 represent neccessary conditions on topological relation between two geometries in order to answer an OP containment. In order to $OP(x, a) \sqsubseteq OP(x, b)$, the relation between the features/geometries a and b has to be according to Table 4.

Then we can define an *effective topological relation restriction* of a geometry individual:

$$r_e(i, q_2) = \begin{cases} 0 \text{ when } |Rel(i, q_2)| > 1, \\ \text{the only element in } Rel(i, q_2) \text{ otherwise.} \end{cases}$$

Using the topological relation restriction, we can define geometry containment relation:

$$x \sim_r y = \begin{cases} x \equiv y & \text{when} \quad r = 0, \\ x \subseteq y & \text{when} \quad r < 0, \\ y \subseteq x & \text{when} \quad r > 0, \end{cases}$$

where the relation $\subseteq$ between two geometries represents that one geometry is a subset (is within) another geometry and the relation $\equiv$ represents that the two geometries are the same.

Then we can extend completed ABox of a data property as:

$$\begin{aligned} &\text{Com}^2(hS(i_x, d_y)) = \text{Com}(hS(i_x, d_y)) \cup \\ &\cup \{(i_x, g) : g \in D_1 \setminus D_{V1} \wedge (d_y \sim_{r_e(i_x, q_2)} g)\} \end{aligned}$$

where $hS \sqsubseteq$ `geo:hasSerialization`, and the rest of the symbols is defined in Section 4.2. When

Egenhofer OPs	$TR(OP)$
`geo:ehEquals`	0
`geo:ehOverlap`	0
`geo:ehDisjoint`	-1
`geo:ehContains`	-1
`geo:ehCoveredBy`	0
`geo:ehCovers`	0
`geo:ehInside`	1
`geo:ehMeet`	0
Simple Feature OPs	**$TR(OP)$**
`geo:sfEquals`	0
`geo:sfIntersects`	0
`geo:sfDisjoint`	-1
`geo:sfContains`	-1
`geo:sfCrosses`	0
`geo:sfTouches`	0
`geo:sfWithin`	1
`geo:sfOverlaps`	0
RCC8 OPs	**$TR(OP)$**
`geo:rcc8eq`	0
`geo:rcc8po`	0
`geo:rcc8dc`	-1
`geo:rcc8ec`	0
`geo:rcc8ntpp`	1
`geo:rcc8ntppi`	-1
`geo:rcc8tpp`	0
`geo:rcc8tppi`	0

Table 3: Values of restrictions $TR(OP)$ of topological relations.

$TR(OP)$	condition
0	$a \equiv b$
1	$a \subseteq b$
-1	$b \subseteq a$

Table 4: Meanings of $TR(OP)$ values.

$\text{Com}^2(DP)$ is used for obtaining $\bar{\mathcal{O}}(a)$ instead of $\text{Com}(DP)$, the query containment decision procedure is extended with GeoSPARQL topological relations reasoning.

This way, containment on even complex query patterns is answered correctly. To give an intuitive proof, we will continue the intuitive proof at the end of Section 4.2. Here the $\text{Com}(DP)$, containing possible substitutions for variables, needs to be extended with substitutions also for geometry literals. But how to select, which geometry literals to substitute? It depends on how the geometry restricts the rest of the query, e.g. when an object has to be within a geometry, the geometry can be substituted with a geometry covering it. The impact spatial restrictions have on geometry substitutions is

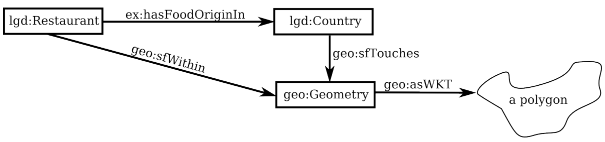


Figure 3: Example of a circle in a query.

given by $TR(OP)$; for a specific object it is computed by $Rel(i, q_2)$, and the effect the spatial restrictions have on the substitution is given by $r_e(i, q_2)$. Then, by comparing geometries based on $r_e(i, q_2)$, it can be correctly decided which geometries to substitute for a geometry in a query to decide query containment with spatial semantics.

Imagine the two queries in Listing 1, structure of which is displayed in Fig. 3. Note that the two polygons there are for example the areas of the Czech Republic (CZ, in both queries) and Slovakia (SK, in q_2 only); the two countries are neighbours.

Listing 1: q_1 and q_2 in circular spatial restriction example.

```
SELECT ?x WHERE {
    ?x a lgd:Restaurant .
    ?x geo:sfWithin ?g .
    ?x ex:hasFoodOriginIn ?c .
    ?c a lgd:Country .
    ?c geo:sfTouches ?g .
q1: ?g geo:asWKT "POLYGON((<CZ>))"
          ^^geo:wktLiteral .
q2: ?g geo:asWKT "POLYGON((<CZ+SK>))"
          ^^geo:wktLiteral .
}
```

Obviously, $q_1 \sqsubseteq q_2$ cannot be true, since the q_1 results contain restaurants in the Czech Republic with their food origin in Slovakia, while the results of q_2 do not. The containment would be true, if there would be only the `geo:sfWithin` spatial restriction.

To show how the proposed reasoning would reach the correct decision, let us show the steps:

$$Rel(i_g, q_2) = \{TR(\texttt{geo:sfTouches}), TR(\texttt{geo:sfWithin})\} = \{0, 1\}$$
$$r_e(i_g, q_2) = 0$$

Therefore, the completed ABox for `geo:asWKT` (¡CZ+SK¿ in q_2) will not be extended with ¡CZ¿ from q_1, thus testing consistency on $\bar{\mathcal{O}}$ of this data property will give consistent, meaning that $q_1 \not\sqsubseteq q_2$.

5 Prototypical Query Lattice Construction and Searching

A lattice is a natural structure for representing a set of queries ordered by their containment, as the set is a partially ordered set (every two queries are related in exactly one of three ways: $q_1 \sqsubseteq q_2$, $q_2 \sqsubseteq q_1$, or q_1 and q_2 are unrelated). Section 5.1 describes how to generate a lattice from a set of queries, Section 5.2 gives details on how to search one.

5.1 Building lattice

All queries form a lattice structure, as any partially ordered set: two queries can be ordered, or can be incomparable. It has the least element, the query giving no results, and the greatest element, the query giving all results.

Algorithm 1 with support functions in Algorithm 2 iteratively builds a lattice, where nodes represent sets of semantically equivalent queries. Each node has a set of data sources capable of answering the node's queries associated to it.

It starts with the lattice being only one root node, representing the query with empty answer (as the top node of the lattice). It adds queries one by one, placing it into appropriate position of the lattice: if the added query *contains* a query (it is lower), but none of its children, add it as a child, as in Algorithm 1, line 28; if it is semantically equivalent to a query, unify it, as in Algorithm 1, line 10; otherwise work recursively, as in Algorithm 1, line 18. The function call initially starts with the inserted query and the root (the no-answer query) as arguments.

5.2 Searching lattice

Algorithm 3 contains the algorithm for searching a user's query in the lattice constructed in Section 5.1. The function is called with the query searched for and the root of the lattice (the no-answer query) as the arguments, then it recursively searches the lattice. It returns all query nodes, which are equivalent to the user's query, or are the "directly" contained ones (which are contained in the user's query, but no other contained in the user's query contains them). In case the algorithm cannot recursively continue at the root level, which means that the user's query does not contain any of the children of root (and hence it would not contain any query in the lattice), it switches to the "splitting" mode.

When in "splitting" mode, the user's query is split to subqueries, which are individually searched in the lattice.

There exist many strategies how to split a query to subqueries in order to find sources capable of answering

Algorithm 1 Lattice construction algorithm

```
Require: r ⊑ q
function InsertIntoLattice(q, r)
    if q ∈ children(r) then
        return
    end if
    doAdd ← true
    inserted ← false
    for all c ∈ children(r) do
        if q ⊑ c then                                ▷ r ⊑ q ⊑ c
            if c ⊑ q then                            ▷ r ⊑ q ≡ c
                unify(c, q)
                return
            end if
            children(r) ← children(r) \ {c}
            children(q) ← children(q) ∪ {c}
            inserted ← true
            break
        else if c ⊑ q then                           ▷ r ⊑ c ⊑ q
            InsertIntoLattice(q, c)
            doAdd ← false
        end if
    end for
    if doAdd then
        if not inserted then
            for all c ∈ children(r) do
                ConnectToChildren(q, c)
            end for
        end if
        children(r) ← children(r) ∪ {q}
    end if
end function
```

Algorithm 2 Support functions for lattice construction algorithm

```
Require: q not comparable to r
function ConnectToChildren(q, r)
    for all c ∈ children(r) do
        if q ⊑ c then                                ▷ q ⊑ c
            if not ChildrenContain(q, c) then
                for all x ∈ children(q) do
                    if ChildrenContain(c, x) then
                        children(q) ← children(q) \ {x}
                    end if
                end for
                children(q) ← children(q) ∪ {c}
            end if
        else
            ConnectToChildren(q, c)                  ▷ q not comparable to c
        end if
    end for
end function
function ChildrenContain(r, x)
    return ⋁_{c∈children(r)} ((x = c) ∨ ∨ChildrenContain(c, x))
end function
```

Algorithm 3 Lattice searching algorithm

```
Require: r ⊑ q
function SearchLattice(q, r)
    S ← {c ∈ children(r) : c ⊑ q}                    ▷ r ⊑ c ⊑ q
    if S = ∅ then
        if r = root of the lattice then
            return UseSplitting(q, r)
        else
            return {(q, r)}
        end if
    else
        return {SearchLattice(q, c) : c ∈ S}
    end if
end function
```

them. It is a compromise between how many sources must be involved in answering the user's query (which includes how much data must be transferred from the sources to the broker and how much processing the broker has to do to combine them) and the extensiveness of the answer (the found query is always contained within the user's query, but the query formed by combining queries from multiple sources may be more fitting the user's query than from an individual source).

One decision is when to do the splitting of the user's query, another one is when to try to join the subqueries back, when some of them can be answered by a single source. The decision may be complex with different optimizations, and it is part of our future work.

Currently, we propose a simple approach to split the user's query at the first level (the children of the lattice root), and then join those splitted subqueries which contain the same child of the root, see Algorithm 4. This reduces both the number of query containment decisions and the number of produced subqueries (i.e. sources necessary to use for the complete answer).

Note that the auxiliary function "join" of a set of subqueries simply returns a new query built from all the subqueries joined (the union of their triples) with the output variables being the union of the queries' output variables. The function "vars" returns a set of all variables appearing in a query or a triple. By the symbol $|q|$ we denote the number of triples in a query q. The symbol $q_1 \subseteq q_2$ represents the syntactic containment, i.e. whether the triples of q_1 are a subset of the triples of q_2 and similarly for their output variables.

Algorithm 4 Trying to split in lattice searching

Require: $r \sqsubseteq q$

function USESPLITTING(q, r)

$X \leftarrow \{(c, s_j) : c \in \text{children}(r) \wedge s_j = \text{join}(\{s \in \text{SPLIT}(q) : c \sqsubseteq s\})\}$ $\triangleright\ r \sqsubseteq c \sqsubseteq s$

return $\{\text{SEARCHLATTICE}(s_j, c) : (c, s_j) \in X \wedge s_j \notin \text{PRUNE}(\{s_j : (c, s_j) \in X\})\}$

end function

The algorithm for the splitting of a query in Algorithm 5 splits the query to a set of queries formed by disjoint triple sets (subsets of the triples of the original query). It splits to the maximum number of subqueries, keeping that for every triple in an output subquery it is true, that all its neighbours via a non-output variable are in the same subquery. To illustrate this, see Fig. 4, where the input query into the algorithm is represented by all triples in the figure, while the triples of the three output queries are represented by the encircled regions A, B, and C.

In theory, a query could be split into single triples, and the broker could let those triples be answered individually by different sources; then the broker would have to do the conjunction itself, requiring that the individuals across different sources match – even the ones representing the user query's non-output variables. As we consider posing less requirements about the alignment between individuals in different sources as good practice, we proposed the algorithm to limit the splitting just to require alignment on the output variables; it is also one way how to limit the number of queries to ask and to reduce the computational demands on the broker. But it is a matter of design choice.

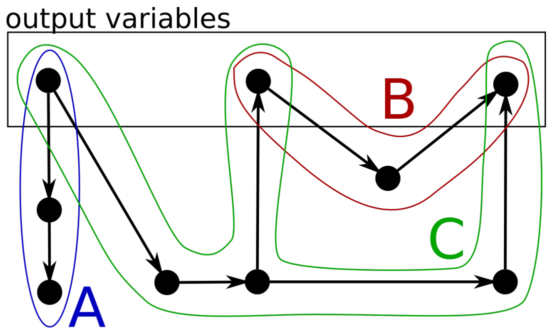


Figure 4: Splitting a query into subqueries. The dots represent distinct variables, the arrows represent predicates.

Algorithm 5 Query splitting

function SPLIT(q)

$T \leftarrow$ all triples of q

$v_o \leftarrow$ output variables of q

$O \leftarrow \emptyset$

while $T \neq \emptyset$ **do**

$t \leftarrow$ one of $\{t \in T : \text{vars}(t) \cap v_o \neq \emptyset\}$

$S \leftarrow \text{SPREAD}(t, T \setminus \{t\}, v_o)$

$T \leftarrow T \setminus S$

$O \leftarrow O \cup \{$a query with triples S and output variables $\text{vars}(S) \cap v_o\}$

end while

return O

end function

function SPREAD(t, T, v_o)

$X \leftarrow \{x \in T : \text{vars}(t) \setminus v_o \cap \text{vars}(x) \neq \emptyset\}$

return $\{t\} \cup \bigcup_{x \in X} \text{SPREAD}(x, T \setminus X, v_o)$

end function

Algorithm 6 is another example of heuristics how to reduce the amount of queries to ask the sources. First, it finds all query subsets which are redundant (note that $\mathcal{P}$ denotes a power set). Next comes the heuristics. From these redundant subsets, we pick only the ones with maximum number of elements (line 3), from these we pick only the ones with the minimum total number of triples with the intent of pruning the least restrictive queries (line 4). When there is more than one such subset available, we pick randomly.

Algorithm 6 Prunning queries

1: **function** PRUNE(Q)
2: $P \leftarrow \{p \in \mathcal{P}(Q) : \text{join}(p) \subseteq \text{join}(Q \setminus p)\}$
3: $P_1 \leftarrow \{p \in P : |p| = \max(\{|p| : p \in P\})$
4: $P_2 \leftarrow \{p \in P_1 : \sum_{q \in p} |q| = \min(\{\sum_{q \in p} |q| : p \in P_1\})$
5: **return** random element of P_2
6: **end function**

6 EXAMPLE

Here is an example, how the lattice construction and searching works. Consider the following background ontology $\mathcal{O}$ consisting of:

- LinkedGeoData ontology [16] (prefix `lgd:`), which describes OpenStreetMap data. Among many others, it contains classes `lgd:Restaurant`, `lgd:Gym`, and `lgd:HistoricBuilding`, which are relevant to our example. It also contains the axiom `lgd:HistoricBuilding` $\sqsubseteq$ `lgd:Historic`.
- GeoNames[13] ontology [18] (prefix `gn:`), describing its own well-classified database of points. It is linked to LinkedGeoData. A little conversion was performed to make it suitable (convert individuals to classes). The following classes are relevant: `gn:S.REST` (restaurants) and `gn:S.HSTS` (historical sites). It also contains axioms that `lgd:Restaurant` $\equiv$ `gn:S.REST` and `lgd:Historic` $\equiv$ `gn:S.HSTS`
- Our example ontology for gourmets (prefix `ex:`). It only contains one object property, `ex:hasFoodOrigin`.

Note that for classes, $C \equiv D$ is only syntactic sugar for $C \sqsubseteq D$ and $D \sqsubseteq C$.

We will consider the following (rather limited, not to clutter our example) geospatial sources (services providing geospatial data, e.g. WFS servers, (Geo)SPARQL endpoints, etc.):

- s_1, having OpenStreetMap data, capable of answering queries with the following restrictions:
  - `lgd:Restaurant`$(?x)$
  - `lgd:Gym`$(?x)$
  - `lgd:Gym`$(?x)$, `lgd:HistoricBuilding`$(?x)$
  - `lgd:HistoricBuilding`$(?x)$
- s_2, having GeoNames data, capable of answering queries with the following restrictions:
  - `gn:S.REST`$(?x)$
  - `gn:S.REST`$(?x)$, `gn:S.HSTS`$(?x)$
  - `gn:S.HSTS`$(?x)$
- s_3, having example gourmet data, capable of answering queries with the following restrictions:
  - `ex:hasFoodOrigin`$(?x, ?c)$
  - `ex:hasFoodOrigin`$(?x, ?c)$, `lgd:Restaurant`$(?x)$

When OnGIS is loaded with the background ontology $\mathcal{O}$ and the sources s_1, s_2, s_3, it creates the lattice in Fig. 5. Note that the root node *bottom* represent the no-answer query, and all the prefixes are omitted for compactness. The abbreviation *R* stands for `Restaurant`, *G* for `Gym`, *HB* for `HistoricBuilding`, and *FO* for `hasFoodOrigin`. As it is a lattice, all nodes with no children depicted should be connected to the *top* node at the bottom, representing the all-answer query. But it has been skipped to make the figure simpler, and neither the OnGIS algorithms operates with the *top* node.

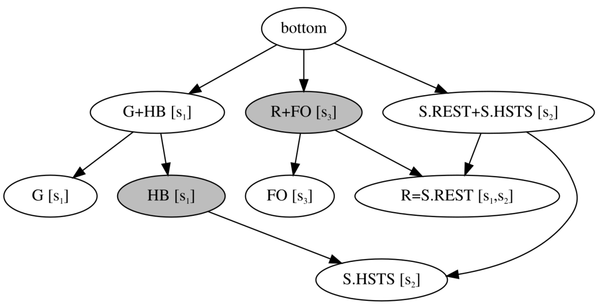


Figure 5: The lattice of example sources.

Now when a user asks the query in Listing 2, it searches the lattice. The algorithm cannot find a single source to answer the complete query, hence it splits the query, and comes up with two partial queries – one for s_1 in Listing 3 and one for s_3 in Listing 4. The lattice nodes used for deciding which sources to use are darker in Fig. 5.

[13] `http://www.geonames.org/`, cit. 27.2.2016.

Listing 2: Example user's query.

```
SELECT ?x ?c ?g WHERE {
   ?x a lgd:Restaurant.
   ?x a lgd:HistoricBuilding.
   ?x ex:hasFoodOrigin ?c.
   ?x geo:hasGeometry
        [geo:hasSerialization ?g].
}
```

Listing 3: Example result query for s_1.

```
SELECT ?x ?g WHERE {
   ?x a lgd:HistoricBuilding.
   ?x geo:hasGeometry
        [geo:hasSerialization ?g].
}
```

Listing 4: Example result query for s_3.

```
SELECT ?x ?c ?g WHERE {
   ?x a lgd:Restaurant.
   ?x ex:hasFoodOrigin ?c.
   ?x geo:hasGeometry
        [geo:hasSerialization ?g].
}
```

7 Conclusion and Future Work

In OnGIS, we created a framework proposal for efficient querying multiple heterogeneous geospatial sources: one part is describing them by sets of prototypical queries the sources can answer, another part is the method how to find which sources to use for answering a user's query (or for answering which part of the query, in case no single source can answer the query entirely). We used GeoSPARQL, a modern geospatial query language, enhanced with OWL 2 QL semantics, for describing the queries. The structure used for searching a user's query is a lattice built from the sources' prototypical queries ordered by semantic query containment.

The proposed algorithms are implemented in a prototype, with successful tests on a few nonlarge samples. There are several ways to make OnGIS better usable in real world scenarios with large data:

- Explore options how to parallelise the lattice construction and searching algorithms.
- Compare two ways of the lattice ordering. The one discussed in this article is "from bottom", where the root is the no-answer query. Another one is "from top", where the root is the all-answer query. We have implemented a prototype of the latter one as well, but some construction and searching operations are more complex than in case of the former one, and our experiments suggest the latter one is slower. But this may be strongly dependent on the characteristics of input data.
- Reduce the amount of prototypical queries for large data sources. One way could be developing a template language for the GeoSPARQL queries to get around the need to list each possible combination of (class, spatial) restrictions as a prototypical query.

Once these options are tackled, OnGIS can be coupled with a geoprocessing component, which would combine partial responses from selected sources (when multiple had to be used for a split user's query), to complete the user's query. This would complete the OnGIS infrastructure to fully functional semantic geospatial data federation system with real-world sources.

References

[1] A. Artale, D. Calvanese, R. Kontchakov, and M. Zakharyaschev, "The DL-Lite family and relations," *J. of Artificial Intelligence Research*, vol. 36, pp. 1–69, 2009.

[2] F. Baader, D. Calvanese, D. L. McGuinness, P. Patel-Schneider, and D. Nardi, *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, 2003.

[3] R. Battle and D. Kolas, "Enabling the geospatial semantic web with Parliament and GeoSPARQL," *Semantic Web Journal*, 2012.

[4] A. Buccella, A. Cechich, and P. Fillottrani, "Ontology-driven geographic information integration: A survey of current approaches," *Computers & Geosciences*, vol. 35, no. 4, pp. 710 – 723, 2009.

[5] E. Clementini, P. Felice, and P. Oosterom, "A small set of formal topological relationships suitable for end-user interaction," in *Advances in Spatial Databases*, ser. Lecture Notes in Computer Science, D. Abel and B. Chin Ooi, Eds. Springer Berlin Heidelberg, 1993, vol. 692, pp. 277–295. [Online]. Available: http://dx.doi.org/10.1007/3-540-56869-7_16

[6] V. Haarslev, R. Moeller, and M. Wessel, *Racer*, 2016, accessed on 21.6.2016. [Online]. Available: https://www.ifis.uni-luebeck.de/index.php?id=385

[7] I. Horrocks, U. Sattler, S. Tessaris, and S. Tobies, "How to decide query containment under constraints using a description logic," in *Logic for Programming and Automated Reasoning*, ser. Lecture Notes in Artificial

Intelligence, M. Parigot and A. Voronkov, Eds. Springer Berlin Heidelberg, 2000, vol. 1955, pp. 326–343. [Online]. Available: http://dx.doi.org/10.1007/3-540-44404-1_21

[8] I. Horrocks, S. Tessaris, U. Sattler, R. Aachen, S. Tobies, R. Aachen, I. Horrocks, S. Tessaris, U. Sattler, S. Tobies, and R. Aachen, "Query containment using a DLR ABox," in *Ltcs-report LTCS-99-15, LuFG Theoretical Computer Science, RWTH*, 1999.

[9] M. Lutz and D. Kolas, "Rule-based discovery in spatial data infrastructure," *Transactions in GIS*, vol. 11, no. 3, pp. 317–336, 2007. [Online]. Available: http://dx.doi.org/10.1111/j.1467-9671.2007.01048.x

[10] Open Geospatial Consortium, *OpenGIS Web Feature Service 2.0 Interface Standard*, 2010, accessed on 21.6.2016. [Online]. Available: http://www.opengeospatial.org/standards/wfs

[11] Open Geospatial Consortium, *OGC GeoSPARQL — A Geographic Query Language for RDF Data*, 2012. [Online]. Available: http://www.opengeospatial.org/standards/geosparql

[12] Open Geospatial Consortium, *OGC WCS 2.0 Interface Standard – Core*, 2012, accessed on 21.6.2016. [Online]. Available: http://www.opengeospatial.org/standards/wcs

[13] E. Sirin and B. Parsia, "SPARQL-DL: SPARQL query for OWL-DL," in *OWLED*, ser. CEUR Workshop Proceedings, C. Golbreich, A. Kalyanpur, and B. Parsia, Eds., vol. 258. CEUR-WS.org, 2007.

[14] M. Šmíd and Z. Kouba, "OnGIS: Methods of searching in spatial data driven by ontologies," in *Digitální technologie v geoinformatice, kartografii a DPZ*, 2012, pp. 107–116.

[15] M. Šmíd and Z. Kouba, "OnGIS: Ontology driven geospatial search and integration," in *Terra Cognita 2012 Workshop*, 2012, pp. 27–38.

[16] C. Stadler, J. Lehmann, and S. Auer, *LinkedGeoData Ontology*, University of Leipzig, accessed on 6.4.2011. [Online]. Available: http://linkedgeodata.org/ontology/

[17] D. Tsarkov and I. Horrocks, *FaCT++*, 2007, accessed on 21.6.2016. [Online]. Available: http://owl.man.ac.uk/factplusplus/

[18] B. Vatant, *The GeoNames Ontology*, GeoNames, 2012, accessed on 27.2.2016. [Online]. Available: http://www.geonames.org/ontology/documentation.html

[19] U. Visser, *Intelligent information integration for the Semantic Web*. Springer, 2005, vol. 3159.

[20] U. Visser, H. Stuckenschmidt, and C. Schlieder, "Interoperability in gis-enabling technologies," in *Proceedings of the 5th AGILE Conference on Geographic Information Science*. Citeseer, 2002, p. 291.

[21] W3C, *SPARQL Query Language for RDF*, 2008, accessed on 21.6.2016. [Online]. Available: https://www.w3.org/TR/rdf-sparql-query/

[22] W3C, *RIF Overview (Second Edition)*, 2013, accessed on 21.6.2016. [Online]. Available: http://www.w3.org/TR/rif-overview/

[23] W3C, *Resource Description Framework (RDF)*, 2014, accessed on 21.6.2016. [Online]. Available: http://www.w3.org/RDF/

[24] M. Wessel and R. Möller, "Flexible software architectures for ontology-based information systems," *Journal of Applied Logic – Special Issue on Empirically Successful Computerized Reasoning*, 2009.

[25] World Wide Web Consortium, *OWL 2 Web Ontology Language: Profiles, OWL 2 QL*, 2009. [Online]. Available: http://www.w3.org/TR/owl2-profiles/#OWL_2_QL

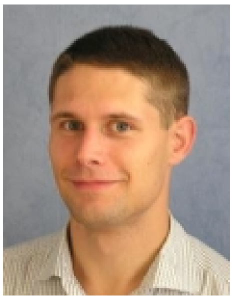
[26] C. Zhang, T. Zhao, and W. Li, "The framework of a geospatial semantic web-based spatial decision support system for digital earth," *Int. J. Digital Earth*, vol. 3, no. 2, pp. 111–134, 2010.

[27] Y. Zhang, Y.-Y. Chiang, P. Szekely, and C. A. Knoblock, "A semantic approach to retrieving, linking, and integrating heterogeneous geospatial data," in *Joint Proceedings of the Workshop on AI Problems and Approaches for Intelligent Environments and Workshop on Semantic Cities*. ACM, 2013, pp. 31–37.

[28] T. Zhao, C. Zhang, M. Wei, and Z.-R. Peng, "Ontology-based geospatial data query and integration," in *GIScience*, ser. Lecture Notes in Computer Science, vol. 5266. Springer, 2008, pp. 370–392.

AUTHOR BIOGRAPHIES

Marek Šmíd is a Ph.D. student in the field of artifical intelligence and biocybernetics from the Czech Technical University in Prague, Czech Republic. His specialization includes semantic technologies, focusing on the language OWL 2 QL, and geographic information systems (GIS). His dissertation thesis deals with GIS data integration using semantic techniques. He took part in projects Netcarity, funded by European Community, and Mondis, supported by Czech Ministry of Culture.



Dr. Petr Křemen received his Ph.D. degree in artifical intelligence and biocybernetics from the Czech Technical University in Prague, Czech Republic. He leads a research team at the Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University, Prague in the field of ontology-based information systems, ontology development, ontology comparison, error explanation and query answering. He is an author of more than 30 peer-reviewed articles, mainly on international fora.