View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by RonPub -- Research Online Publishing

© 2016 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 3, Issue 1, 2016
http://www.ronpub.com/ojsw
ISSN 2199-336X

OnGIS: Semantic Query Broker for Heterogeneous Geospatial Data Sources

Marek Šmíd, Petr Křemen

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 in the data integration and in the subsequent result interpretation. This paper proposes broker techniques for answering a user's complex spatial query: finding relevant data sources (from a catalogue of data sources) capable of answering the query, eventually splitting the query and finding relevant data 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, heterogeneous data, lattice, OnGIS*

1 Introduction

OnGIS is a semantic geospatial query broker. Some parts of it (simple semantic data integration, two options of a user interface) have been previously developed as prototypes, see [28, 29].

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 achieved the two goals by describing the sources with prototypical queries that the sources can answer (a set of prototypical queries for each source). A prototypical query is a semantic geospatial query capturing capabilities of what a source can answer, where free variables can be placed where the source can answer the query with any value in the variable's place, e.g. a geometry for a spatial restriction, or a literal for a data property restriction.

Prototypical queries fully capture relevant capabilities of a data source in terms of data and operations the source provides. It is necessary since heterogeneous sources and services are considered, and therefore no single specific technology for describing capabilities can be used. Such examples include: (a) Traditional GIS web services, e.g. Open Geospatial Consortium (OGC)[1], a large consortium, is setting many standards, e.g. Web Feature Service (WFS) [22] that is a standard for publishing vector spatial data, Web Coverage Service (WCS) [24] that is a standard for publishing source raster data, ESRI ArcGIS services [12], etc.; (b) Plain relational databases, e.g. PostgreSQL+PostGIS, Oracle

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

Spatial, etc.; (c) Linked Data[2] [39] sources available through SPARQL [36] endpoints, e.g. DBpedia [2], LinkedGeoData [31], etc. Each can serve different data and have different capabilities.

Linked Data is a promising method for publishing and integrating different kinds of data including spatial data. For example, OpenStreetMap data have already been 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. SPARQL is a query language for data in Resource Description Framework (RDF) [38], and RDF is a standard model for data interchange and it 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, forms a special data structure with them. When a user asks a query, the system 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 (a method stating the relation of two queries; for definition see Section 4), 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 [28, 29], and the work of others (see Section 2), OnGIS supports a semantic discovery of geospatial sources capable of answering parts of a query.

The rest of this paper is structured as follows. In Section 2 we briefly explore existing methods and technologies related to our work. Section 3 gives background on GeoSPARQL and OWL 2 QL. Section 4 contains all necessary parts of the query containment we have designed, inlcuding the computation of the query containment and the GeoSPARQL semantics. Section 5 describes how to construct and search a lattice of queries. Section 6 evaluates our implmentation. Finally, Section 7 concludes the text.

2 Related Work

There are some techniques for defining and searching geographic information system (GIS) catalogues. For example, Catalogue Service (CSW) [25], a standard by Open Geospatial Consortium, is an interface to discover, browse, and query metadata about GIS data and services. CSW uses Dublin Core[4] vocabulary to describe web resources, which can be searched by metadata (keywords, author, date, etc). However, CSW does not allow for more complex queries or semantic search and for spatial querying only bounding box is supported, since the the capability of CSW for for describing sources is very limited.

There is also some work on using semantic technologies for spatial data. In [40], an ontology-based information system is implemented, focusing on ontology-based spatio-thematic query answering for city maps. The system bases on description logics reasoner RacerPro [15], which implements a more expressive logic (compared to what we use) $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D}^-)$. Note that these letters denote what constructs are allowed in the description logics that RacerPro uses. This notation is used throughout this paper, and for its explanation see [3]. The system in [40] implements its custom storage, which directly includes the inference algorithms and the query evaluation engine. A custom query language SuQL (the substrate query language, also in [40]) is used. However, [40] does not solve the problem of integrating multiple data sources.

The authors in [4] use Parliament triple store for supporting geospatial indexes, for storing spatial data and for making complex spatial queries via GeoSPARQL. However, [4] uses a precomputed data set and does not directly support data integration.

The system in [45] links an RDF ontology to databases and WFS [22]. 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 [43] integrates various data sources (e.g. 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.

The work in [21] has a similar goal as our OnGIS. [21] proposes an interesting system, which is also based on semantic technologies. But instead of open-world OWL semantics, the work uses rules (specifically SWRL rules) for integrating sources and for answering queries. The problem of data integration is also summarized in [21].

Buster [35, 34] 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

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

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

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

intelligent information broker. Buster represents its terminology using the language OIL [13] and the description logic $\mathcal{SHIQ}$ and uses Dublin Core as the vocabulary for modeling metadata. Buster 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 the participation of multiple sources on one user's query.

The system Karma presented in [44] 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, Karma uses only limited RDF expressive power and does not support complex spatial queries.

A comprehensive overview of related work in the area of ontology-driven GIS integration is in [6].

As query containment is an important part of our query-broking solution (see Section 4), the capabilities of query containment of several systems are examined here. FaCT++ [32] and its predecessor FaCT [16] are reasoners supposed to support query containment, but they do not support custom datatypes (which are necessary for GeoSPARQL). Unfortunately, the description of how their query containment works could not be found, even digging in their C++, respectively LISP source codes did not help.

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 [27], 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 that query containment 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 to a resource (IRI) and a literal. This means the property would be both data property and object property, and this is illegal in OWL.

In [7], the author shows a method to decide query containment on SPARQL with OWL 2 EL [41] ontologies using the translation into μ-calculus [5]. OWL 2 EL is of polynomial data complexity, which is higher than AC^0 complexity of OWL 2 QL (which allows data query answering be performed directly by relational databases). The method used is complex enough to support property paths and optional graph patterns, and to cover other OWL 2 profiles as well. For example, the author states that the method of quey containment in [7] can be used for OWL 2 QL without inverse roles.

In [26], the authors deeply analyze the complexity of query containment over "well-designed" SPARQL queries supporting optional graph patterns and unions, without RDFS or any other entailment regimes. The results are however not reusable for our case, as we leverage an entailment regime – the OWL 2 QL reasoning.

In [10], there is an evaluation of three SPARQL query containment solvers, two supporting RDFS, one not. Two of them are actually μ-calculus containment solvers and need some translation.

In [9] and [8], the authors propose methods for SPARQL query containment in $\mathcal{SHI}$ description logics, respectively under RDFS entailment regime. Both methods use a translation into μ-calculus.

The methods for deciding query containment, which are based on μ-calculus and presented in [7, 9, 8], should be possible to extend for OWL 2 QL semantics, and probably could also be extended with geospatial reasoning. It would be interesting to see the result and compare their performances to our method for deciding query containment.

Good ways how to optimize querying multiple sources are in [14], where the authors nicely summarize existing and present new ways how to perform efficient SPARQL query federation. The techniques presented there could be used to optimize OnGIS in the future.

An alternative to GeoSPARQL is stSPARQL [19], which is another query language based on SPARQL with spatial and temporal extension functions. Most features of the two query languages are similar with some differences: GeoSPARQL has three families of topological relations modeled both as properties and functions (see Section 3.1), while stSPARQL has one family of topological relations as functions only. On the other hand, stSPARQL has temporal functions, geospatial aggregate functions, and functions for minimum bounding boxes (none of these are necessary for the current version of OnGIS). stSPARQL has been developed as part of Strabon [20], a semantic spatiotemporal RDF store. The main

[5] `https://github.com/Complexible/pellet`, cit. 7.8.2016.

[6] `http://ontop.inf.unibz.it/`, cit. 4.6.2014.

[7] `http://spinrdf.org/`, cit. 10.5.2014.

[8] `http://spinrdf.org/sp#object`

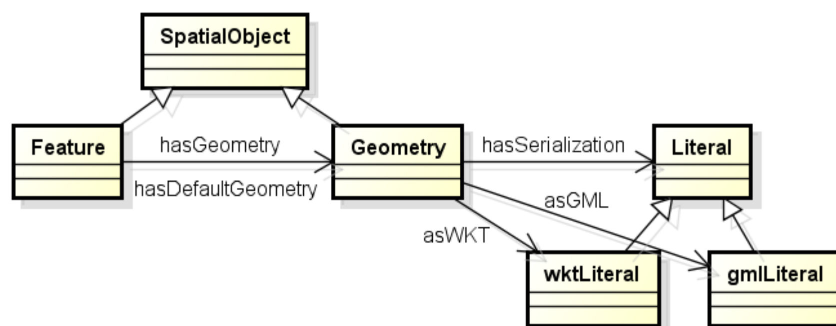


Figure 1: GeoSPARQL basic classes and properties

reason GeoSPARQL has been selected for representing semantic geospatial queries is its chance to become widely used, as it is defined by the well-known OGC.

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 the containment of GeoSPARQL queries, we need a technique requiring the logical negation, and the logical negation 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 for our purposes), and computational properties (it is tractable, i.e. evaluable using a relational database; any increase in expressive power would break this property). OWL 2 QL is therefore suitable also as the language for querying data from the respective sources.

The translation of OWL 2 QL data queries into SQL is used in our previous work in [28, 29]. There are other profiles of OWL 2, namely EL and RL, but they are not suitable for translation to RDBMS queries (see [42]). OWL 2 EL is more expressive than QL (it supports e.g. negative property assertions, functional data properties), OWL 2 RL uses rule-based reasoning, therefore it does not satisfy open-world assumption.

3.1 GeoSPARQL

GeoSPARQL is a relatively recent standard published by OGC [23]. It extends SPARQL query language for RDF data, adding support for spatial data and spatial operations.

Basic GeoSPARQL concepts are described in Fig. 1. GeoSPARQL contains three basic classes: `SpatialObject`, and its 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 one is for the data themselves, 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 its boundary).

GeoSPARQL defines object properties for topological relations. There are three definition families of the relations, and each contains eight relations. These properties and relations are outlined in Table 1. Every such object property has its domain and range equal to `SpatialObject`.

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

Table 1: Topological relations with their meanings, divided into families

object property	meaning
Simple Feature family	
`sfEquals`	spatially equal
`sfDisjoint`	disjoint (cannot touch)
`sfIntersects`	share at least a point
`sfTouches`	externally touch
`sfWithin`	inside (can touch boundary)
`sfContains`	inverse of `sfWithin`
`sfOverlaps`	some points common, same dimension
`sfCrosses`	e.g. line crosses area
Egenhofer family	
`ehEquals`	spatially equal
`ehDisjoint`	disjoint (cannot touch)
`ehMeet`	externally touch
`ehOverlap`	overlap
`ehCovers`	inverse of `ehCoveredBy`
`ehCoveredBy`	inside (can touch boundary)
`ehInside`	inside (cannot touch boundary)
`ehContains`	inverse of `ehInside`
RCC8 – Region Connection Calculus[9] family	
`rcc8eq`	spatially equal
`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

be described by the DE-9IM model, which uses 3×3 matrices; for details see [11].

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 the 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) [37] 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:`.

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

Table 2: 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^-	$(P^-)^{\mathcal{I}} = = \{(b,a) \mid (a,b) \in P^{\mathcal{I}}\}$	inverse of a role
$\exists R$	$(\exists R)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \exists b : : (a,b) \in R^{\mathcal{I}}\}$	existential quantification
$\neg B$	$(\neg B)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus B^{\mathcal{I}}$	negation of a basic concept

3.2 OWL 2 QL

OWL 2 QL [42] 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 [3] defined in [1]. $DL{-}Lite^{\mathcal{H}}_{core}$ constructs for defining concepts and roles in description logics syntax are:

$$B ::= A \mid \exists R, \;\; C ::= B \mid \neg B, \;\; R ::= P \mid P^-,$$

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, which 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 $\Delta^{\mathcal{I}}$. The semantics of the used constructs are defined in Table 2.

A TBox (a set of all ontology terminological axioms, e.g. subsumptions of concepts, domains, etc.) can be defined by inclusion axioms of the form: $B \sqsubseteq C$, and

[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 3: 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

$R_1 \sqsubseteq R_2$, interpreted by $\mathcal{I}$ as $B^{\mathcal{I}} \subseteq C^{\mathcal{I}}$, respectively $R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$.

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 given in Table 3.

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.

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

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, basic classes, properties, serializations, topological relations, and some other 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, a hierarchy on topological relations is defined, Section 4.2 develops a 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 the semantics of GeoSPARQL is to understand the relations among 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 the containment of queries. For example, both `rcc8tpp` and `rcc8ntpp` (i.e. 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 visualization in Protégé[12] is in Fig. 2.

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

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

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

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 restrictions 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 [18] and [17]: Take 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 $\mathcal{O} \models q_1 \sqsubseteq q_2$. Our modifications of the already proposed methods include adapting them for the OWL 2 QL logics (originally they support $\mathcal{DLR}$ logic, having relations of any arity, by a translation into satisfiability in $\mathcal{SHIQ}$ description logics), and adding support for spatial reasoning by extending completed ABox in Section 4.3.

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 $\mathcal{O}$ for a background ontology.

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

$$\begin{aligned}
&\text{Com}(OP(i_x, i_y), q_1, q_2) =\\
&\quad = \alpha(i_x, i_y, q_1, q_2) \cup \beta(i_x, i_y, q_1, q_2)\\
&\alpha(i_x, i_y, q_1, q_2) =\\
&\quad = \begin{cases} \{(i'_x, i_y) : i'_x \in I_1 \setminus I_{V1}\} & \text{if } i_x \in I_{V2}\\ \emptyset & \text{otherwise}\end{cases}\\
&\beta(i_x, i_y, q_1, q_2) =\\
&\quad = \begin{cases} \{(i_x, i'_y) : i'_y \in I_1 \setminus I_{V1}\} & \text{if } i_y \in I_{V2}\\ \emptyset & \text{otherwise}\end{cases}\\
&\text{Com}(DP(i_x, d_y), q_1, q_2) =\\
&\quad = \gamma(i_x, d_y, q_1, q_2) \cup \delta(i_x, d_y, q_1, q_2)\\
&\gamma(i_x, d_y, q_1, q_2) =\\
&\quad = \begin{cases} \{(i'_x, d_y) : i'_x \in I_1 \setminus I_{V1}\} & \text{if } i_x \in I_{V2}\\ \emptyset & \text{otherwise}\end{cases}\\
&\delta(i_x, d_y, q_1, q_2) =\\
&\quad = \begin{cases} \{(i_x, d'_y) : d'_y \in D_1 \setminus D_{V1}\} & \text{if } d_y \in D_{V2}\\ \emptyset & \text{otherwise}\end{cases}
\end{aligned}$$

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 NOP for each OP and NDP for each DP.

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 [18] and [17], as our steps are based on those articles with suitable modifications and simplifications for OWL 2 QL semantics. Detailed proofs are out of the scope of this text.

Intuitive proof: in order to $q_1 \sqsubseteq q_2$ be valid, q_1 has to restrict its 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. In a 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 Com(OP) and 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 checking in the decisions of query containment, it is necessary to substitute only non-variable individuals/literals in A_1.

To analyze complexity of the decisions, we follow complexity of the steps: The size of Can(q) is linear to the size of q, the size of Com(an assertion axiom, q_1, q_2) is also linear to the size of q_1, and the size of $\bar{\mathcal{O}}(a)$ is linear to the size of $\mathcal{O}$, q_1, and q_2 combined. Therefore, a query containment decision requires approximately $|q_2| \times (|\mathcal{O}| + |q_1| + |q_2|)$ consistency checks in OWL 2 QL, i.e. a polynomial number in the query sizes and the background ontology size. And since consistency checks in OWL 2 QL are of NLogSpace-complete complexity [42], the overall complexity of the query containment is PolyTime. Note that this complexity is lower than the complexity ExpTime in [17], thanks to lower expressive power of OWL 2 QL.

To deal with filters, a different approach has to be used because filters have multiple arity and different semantics (they do not have the open world assumption), thus using their reification, negation, and consistency checks do 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. Here we give only the idea of the algorithm with the details skipped.

We make only one assumption in variable mapping: Output variables are the same. This is a realistic assumption, since GIS systems in simple cases generate e.g. objects' geometries and labels. Therefore, the implemented semantic GIS system may fix constant output variable names and types. Then the variable mapping algorithm recursively searches all possible combinations of mapping variable names from one query to the other, starting from the output variables, preserving one condition: When a subset of variables in one query is grouped together (e.g. by a query triple or by a query filter), it must be grouped in the other query as well.

4.3 Adding Support for GeoSPARQL

When deciding query containment with queries only on the 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

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

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 4. Note that in a topological relation both a geometry individual and a feature individual can play roles (thus the logical *or* in the definition).

Table 4: Values of restrictions $TR(OP)$ of topological relations

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 5: Meanings of $TR(OP)$ values.

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

The numerical values of $TR(OP)$ in Table 4 represent necessary conditions on the 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 5.

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

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

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

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

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 $\text{Com}^2(DP)$ 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. For example, when an object has to be within a geometry, the geometry can be substituted with a geometry covering it. The impact, which spatial restrictions have on geometry substitutions, is given by $TR(OP)$; for a specific object it is computed by $Rel(i, q_2)$, and the effect that 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.

Note that since all steps to obtain Com^2 do not exceed PolyTime complexity, using this spatial extension does not affect the overall complexity of query containment.

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

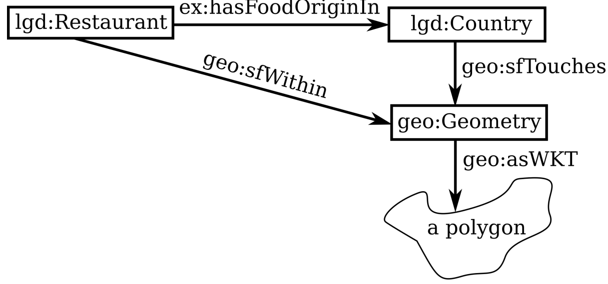


Figure 3: Example of a circle in a query.

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 the lattice for a given query.

5.1 Building Lattice

All queries form a lattice structure as any partially ordered set: Two queries can be ordered, or can be incomparable. A lattice is an algebraic structure, which has the least element and the greatest element[13]. In our case, the least element is the abstract query giving no results and the greatest element is the abstract 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 with it.

Algorithm 1 Lattice construction algorithm

```
Require: r ⊑ q
 1: function InsertIntoLattice(q, r)
 2:     if q ∈ children(r) then
 3:         return
 4:     end if
 5:     doAdd ← true
 6:     inserted ← false
 7:     for all c ∈ children(r) do
 8:         if q ⊑ c then                        ▷ r ⊑ q ⊑ c
 9:             if c ⊑ q then                    ▷ r ⊑ q ≡ c
10:                 unify(c, q)
11:                 return
12:             end if
13:             children(r) ← children(r) \ {c}
14:             children(q) ← children(q) ∪ {c}
15:             inserted ← true
16:             break
17:         else if c ⊑ q then                   ▷ r ⊑ c ⊑ q
18:             InsertIntoLattice(q, c)
19:             doAdd ← false
20:         end if
21:     end for
22:     if doAdd then
23:         if not inserted then
24:             for all c ∈ children(r) do
25:                 ConnectToChildren(q, c)
26:             end for
27:         end if
28:         children(r) ← children(r) ∪ {q}
29:     end if
30: end function
```

[13] For details, see `https://en.wikipedia.org/wiki/Lattice_(order)`, cit. 23.8.2016.

Algorithm 2 Support functions for lattice construction algorithm

Require: q not comparable to r
1: **function** CONNECTTOCHILDREN(q, r)
2:     **for all** $c \in$ children(r) **do**
3:         **if** $q \sqsubseteq c$ **then** ▷ $q \sqsubseteq c$
4:             **if** not CHILDRENCONTAIN(q, c) **then**
5:                 **for all** $x \in$ children(q) **do**
6:                     **if** CHILDRENCONTAIN(c, x) **then**
7:                         children(q) $\leftarrow$ children(q) $\setminus \{x\}$
8:                     **end if**
9:                 **end for**
10:                 children(q) $\leftarrow$ children(q) $\cup \{c\}$
11:             **end if**
12:         **else**
13:             CONNECTTOCHILDREN(q, c) ▷ q not comparable to c
14:         **end if**
15:     **end for**
16: **end function**
17: **function** CHILDRENCONTAIN(r, x)
18:     **return** $\bigvee_{c \in \text{children}(r)} ((x = c) \vee$ CHILDRENCONTAIN$(c, x))$
19: **end function**

Algorithm 1 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 a query into appropriate position of the lattice: If the to be added query *contains* a query in the lattice, but contains 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.

In the worst case, adding a query to the lattice requires the amount of query containment decisions equal to twice the size of the lattice (when caching of query containment results is in place, as the algorithm may encounter the same query pair multiple times), i.e. to build a lattice out of n prototypical queries requires the maximum of $n(n-1)$ query containment decisions. But in practical situations, it is usually less, see Section 6.

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, and this means that the user's query does not contain any of the children of the root (and hence it would not contain any query in the lattice), Algorithm 3 switches to the "splitting" mode, which continues with the Algorithm 4.

Algorithm 3 Lattice searching algorithm

Require: $r \sqsubseteq q$
1: **function** SEARCHLATTICE(q, r)
2:     $S \leftarrow \{c \in \text{children}(r) : c \sqsubseteq q\}$ ▷ $r \sqsubseteq c \sqsubseteq q$
3:     **if** $S = \emptyset$ **then**
4:         **if** $r =$ root of the lattice **then**
5:             **return** USESPLITTING(q, r)
6:         **else**
7:             **return** $\{(q, r)\}$
8:         **end if**
9:     **else**
10:         **return** $\{$SEARCHLATTICE$(q, c) : c \in S\}$
11:     **end if**
12: **end function**

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

Searching a lattice of n nodes should take, in the worst case, n query containment decisions in case the splitting is not used, and $n + n|q|$ query containment decisions when the splitting is used (where $|q|$ is the number of triples in q). This should happen only when the lattice is of a deformed shape (e.g. all nodes are children of the root, or it is a chain). In practical situations, the worst

Algorithm 4 Trying to split in lattice searching

Require: $r \sqsubseteq q$
1: **function** USESPLITTING(q, r)
2: $X \leftarrow \{(c, s_j) : c \in \text{children}(r) \wedge s_j = \text{join}(\{s \in \text{SPLIT}(q) : c \sqsubseteq s\})\}$ ▷ $r \sqsubseteq c \sqsubseteq s$
3: **return** $\{\text{SEARCHLATTICE}(s_j, c) : (c, s_j) \in X \wedge s_j \notin \text{PRUNE}(\{s_j : (c, s_j) \in X\})\}$
4: **end function**

case happens less as showed in Section 6.

There exist many strategies how to split a query to subqueries in order to find sources capable of answering 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 in the users query, but the query formed by combining queries from multiple sources may give more results to the users query than a single query 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 split subqueries which contain the same child of the root, as described in 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. 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.

The algorithm for the splitting of a query in Algorithm 5 splits the query into a set of subqueries, and each subquery is formed by a disjoint subset of triples of the original query. Every such triple subset is selected to be of minimal size, but keeping the condition that for every triple t in the subset, all triples sharing a non-output variable with t are in the same subset (a non-output variable is a variable not appearing in the original query SELECT statement). This is illustrated in Fig. 4, where the query formed by all triples in the figure is the algorithm input, and the three subqueries represented by the triples in the encircled regions A, B, and C are the algorithm output.

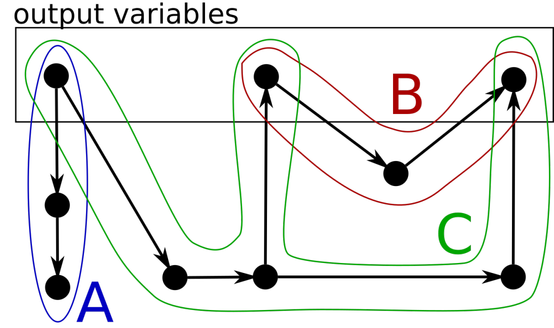


Figure 4: Splitting a query into subqueries (The dots represent distinct variables, the arrows represent predicates)

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 an algorithm, which keeps the originally inner variables not exposed by the split subqueries and just requires alignments on the output variables. It is also one way how to limit the number of queries to reduce the computational demands on the broker. But it is a matter of design choice.

Algorithm 6 is another example of heuristics how to reduce the number of queries necessary to be answered by the sources. Note that by the symbol $|q|$ we denote the number of triples in a query q. First, Algorithm 6 finds all query subsets that are redundant (note that $\mathcal{P}$ denotes a power set). Next, comes the heuristics. From these redundant subsets, we pick only the ones with the maximum number of elements (line 3), from which 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 a subset randomly.

Algorithm 5 Query splitting

1: **function** SPLIT(q)
2: $T \leftarrow$ all triples of q
3: $v_o \leftarrow$ output variables of q
4: $O \leftarrow \emptyset$
5: **while** $T \neq \emptyset$ **do**
6: $t \leftarrow$ one of $\{t \in T : \text{vars}(t) \cap v_o \neq \emptyset\}$
7: $S \leftarrow \text{SPREAD}(t, T \setminus \{t\}, v_o)$
8: $T \leftarrow T \setminus S$
9: $O \leftarrow O \cup \{$a query with triples S and output variables $\text{vars}(S) \cap v_o\}$
10: **end while**
11: **return** O
12: **end function**
13: **function** SPREAD(t, T, v_o)
14: $X \leftarrow \{x \in T : \text{vars}(t) \setminus v_o \cap \text{vars}(x) \neq \emptyset\}$
15: **return** $\{t\} \cup \bigcup_{x \in X} \text{SPREAD}(x, T \setminus X, v_o)$
16: **end function**

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 TESING OF PROTOTYPICAL IMPLEMENTATION

This sections tests our prototypical implementation of the construction and searching of the query lattice by two study cases. Finally, we compare our method of query containment with other implementations. The first testing case does not feature any spatial restriction and uses the splitting of queries, and the second example has spatial restrictions and does not use the query splitting.

6.1 The First Testing Case

Here is the first example of how the lattice construction and searching works. It has been successfully tested using the our prototypical implemenation. Consider the following background ontology $\mathcal{O}$ consisting of:

- LinkedGeoData ontology [30] (prefix `lgd:`), which describes OpenStreetMap data. Among many others, it contains classes `lgd:Restaurant`, `lgd:Gym`, and `lgd:HistoricBuilding`, which are relevant to our testing case. It also contains the axiom `lgd:HistoricBuilding` $\sqsubseteq$ `lgd:Historic`.
- GeoNames[14] ontology [33] (prefix `gn:`), describing its own well-classified database of points. It is linked to LinkedGeoData. A little conversion was performed to make it suitable (converting individuals to classes). The following classes are relevant: `gn:S.REST` (restaurants) and `gn:S.HSTS` (historical sites). Also the following axioms are included: `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 geospatial sources (services providing geospatial data, e.g. WFS servers [22], (Geo)SPARQL endpoints, etc.):

- s_1, having OpenStreetMap data, capable of answering queries with the following restrictions:
  - `lgd:Restaurant`$(?x)$
  - `lgd:Gym`$(?x)$

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

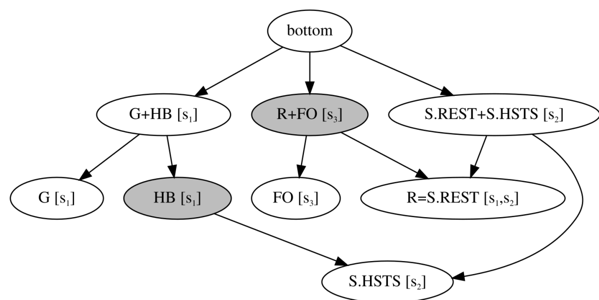


Figure 5: The lattice of example sources.

Table 6: The statistics of the first testing case

	Value/Average	Std. dev.
Background ontology	16,251 axioms	
Prototypical queries	9	
Lattice size	8	
Lattice construction		
Containment decisions	50	
Time	5.01 s	0.62 s
Lattice search		
Containment decisions	20	
Time	214 ms	30 ms

  - `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* represents 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.

Now when a user asks the query in Listing 2, the prototype searches the lattice. It 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.

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].
}
```

Some statistics of the example using our OnGIS prototype are given in Table 6. The response times are computed from 20 lattice constructions and 200 lattice searches on a Linux laptop with Intel Core i7 @ 2.4 GHz with Oracle Java 8 (1.8.0_101) without any parallelization. Notice that the reasoner used for deciding consistency is Pellet (version 2.3.1), which is designed for more expressive logic than OWL 2 QL. Therefore designing reasoner tailored for OWL 2 QL consistency checks gives space for optimizations.

Notice that both numbers of containment decisions are lower than the theoretical maximums, $9(9-1) = 72$ for lattice construction, and $8 + 8 \cdot 5 = 48$ for lattice searching with the query splitting used.

Listing 5: Example of prototypical query in the second example

```
SELECT ?x ?g WHERE {
  ?x a dbp:Restaurant.
  ?x geo:hasGeometry
       [geo:hasSerialization ?g].
  ?x geo:ehInside
       [a sf:Polygon; geo:asWKT
       "POLYGON((−31.3 81,39.9 81...))"
       ^^geo:wktLiteral].
}
```

Listing 6: User's query for the second testing case

```
SELECT ?x ?g WHERE {
  ?x a lgd:Restaurant.
  ?x geo:hasGeometry
       [geo:hasSerialization ?g].
  ?x geo:sfWithin
       [a sf:Polygon; geo:asWKT
       "POLYGON((11 52,20 52,20...))"
       ^^geo:wktLiteral].
}
```

6.2 The Second Testing Case

The second case, also successfully tested in our OnGIS, uses LinkedGeoData, GeoNames, and DBpedia ontologies as the background ontologies. The testing example has three sources, each one using one of the three ontologies, altogether containing more prototypical queries than in the previous example. Moreover, the prototypical queries contain spatial restrictions, representing that the sources contain data only in specified areas. For that purpose, the `geo:ehInside` topological relation is used, together with a polygon serialized by a WKT string. Thus, a prototypical query can look like the one in Listing 5.

The three sources are:

- s_{lgd}, the data source described by LinkedGeoData. The source contains 43 prototypical queries, which all are spatially restricted by a rectangle that bounds the area of Prague (the capital of the Czech Republic),
- s_{gn}, the data source described by GeoNames ontology. The source contains 34 prototypical queries, which all are spatially restricted by a rectangle that bounds the area of the Czech Republic,
- s_{dbp}, the data source described by DBpedia ontology. The source contains 3 prototypical queries, which all are spatially restricted by a rectangle that bounds the area of Europe.

Each of these sources contains a class for restaurants, and these classes are linked together as being equal. So when a user asks the query in Listing 6 (containing rectangular spatial restriction slightly larger than the Czech Republic), OnGIS correctly answers with the source s_{gn}, which is the most fitting one. Even though the other two sources contain equal classes for restaurants, the corresponding prototypical query in s_{lgd} is narrower (strictly contained within the users query) than the selected prototypical query in s_{gn}. The corresponding prototypical query in s_{dbp} is wider than the user's query, and it does not satisfy to be contained within the user's query.

Also note that the user's query uses the topological relation `geo:sfWithin`, while the sources for spatially restricting their data use `geo:ehInside`. Since `geo:ehInside` is a sub-property of `geo:sfWithin` (see Fig. 2), the spatial restriction given by the user is weaker than the spatial restrictions given by the sources, therefore the query can be answered by s_{gn}. When the user's query would use a stricter spatial restriction, e.g. `geo:rcc8ntpp`, none of the sources could satisfy the query.

The resulting statistics of the second testing case is given in Table 7. Again, both numbers of containment decisions are lower than the theoretical maximums: $80(80-1) = 6,320$ for lattice construction, and 80 for lattice searching without the query splitting used.

6.3 Comparison with Other Systems

We tried to compare our proposed method of query containment with other implementations available. As stated in Section 2, using the reasoners FaCT and FaCT++ [32, 16] was not technically possible.

We succeeded using the query containment method in Pellet (Pellet is also used as the reasoner for consistency checks in our prototype), however, with some limitations. A look into the Java class responsible for query containment in Pellet, `QuerySubsumption`[15], reveals it is partially similar to a part of the solution presented in Section 4.2, based on [18, 17]. For deciding whether $\mathcal{O} \models q_1 \sqsubseteq q_2$, Pellet computes $T = \text{Can}(q_1) \cup \mathcal{O}$, and tries to answer q_2 over T. $q_1 \sqsubseteq q_2$ iff the result is not empty. The translation seems simpler compared to our approach, but it is not clear how easy it would be to extend it with spatial reasoning,

[15] Available at `https://github.com/Complexible/pellet/blob/master/query/src/main/java/com/clarkparsia/pellet/sparqldl/engine/QuerySubsumption.java`, cit. 7.8.2016.

Table 7: The statistics of the second tesing case

	Value/Average	Std. dev.
Background ontology	31,331 axioms	
Prototypical queries	80	
Lattice size	80	
Lattice construction		
Containment decisions	5,568	
Time	239.9 s	10.2 s
Lattice search		
Containment decisions	69	
Time	3.66 s	0.23 s

Table 8: The execution times for the simplified first testing case by our versus Pellet's query containment

	Average	Std. dev.
Lattice construction		
Our query containment	713 ms	442 ms
Pellet query containment	1017 ms	659 ms
Lattice search		
Our query containment	136 ms	23 ms
Pellet query containment	146 ms	24 ms

and it uses conjunctive query answering (which is NP-complete in the case of OWL 2 QL), instead of consistency checks (NLogSpace-complete in OWL 2 QL).

As Pellet does not support data properties, we tested it only with a simplified version of our first testing case, where we skipped the `geo:hasGeometry` part for retrieving geometries both in prototypical queries and in the user's query (which would be needed for an actual source querying, but it may be skipped in our example).

We replaced our query containment algorithm with the one Pellet provides and kept the rest of the OnGIS prototype the same (lattice construction and searching). For the first testing cases, Pellet gvies the same results for the lattice construction and search, but it takes more thime than our approach. A comparion of the execution times for the simplified first testing case by our versus Pellet's query containment are in Table 8.

Unfortunately, other query containment decision systems mentioned in Section 2 have not been found or have not been accessible.

7 Conclusion and Future Work

In this paper, we developed a work for efficient querying multiple heterogeneous geospatial sources, and implemented in the system OnGIS. One part of this work is describing heterogeneous geospatial sources by sets of prototypical queries that these sources can answer. Another part is how to find the data sources, which can answer the user's query (or for answering a part of the query, in case no single source can answer the entire query). 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 and successfully tested 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 the 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 sources had to be used for a split user's query) to complete the user's query. This would complete the OnGIS infrastructure and make it a fully functional semantic geospatial data federation system for 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] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "Dbpedia: A nucleus for a web of open data," in *Proceedings of the 6th International The Semantic Web and 2nd Asian Conference on Asian Semantic Web Conference*.

Berlin, Heidelberg: Springer-Verlag, 2007, pp. 722–735.

[3] 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.

[4] R. Battle and D. Kolas, "Enabling the geospatial semantic web with Parliament and GeoSPARQL," *Semantic Web Journal*, vol. 3, no. 4, pp. 355–370, Oct. 2012.

[5] J. Bradfield and C. Stirling, *Handbook of modal logic*. Elsevier, Nov 2006, vol. 3, ch. Modal Mu-Calculi, pp. 721–756.

[6] 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.

[7] M. W. Chekol, "On the Containment of SPARQL Queries under Entailment Regimes," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016, pp. 936–942.

[8] M. W. Chekol, J. Euzenat, P. Genevès, and N. Layaïda, "SPARQL query containment under RDFS entailment regime," in *Proceedings of Automated Reasoning: 6th International Joint Conference*, B. Gramlich, D. Miller, and U. Sattler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 134–148.

[9] M. W. Chekol, J. Euzenat, P. Genevès, and N. Layaïda, "SPARQL query containment under SHI axioms," in *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, ser. AAAI'12. AAAI Press, 2012, pp. 10–16.

[10] M. W. Chekol, J. Euzenat, P. Genevès, and N. Layaïda, "Evaluating and benchmarking SPARQL query containment solvers," in *Proc. 12th International semantic web conference (ISWC)*, vol. 8219. Sydney, Australia: Springer Verlag, Oct. 2013, pp. 408–423. [Online]. Available: https://hal.inria.fr/hal-00917911

[11] 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.

[12] Esri, *ArcGIS for Server – Publish Services*, 2016, accessed on 4.8.2016. [Online]. Available: http://server.arcgis.com/en/server/10.4/publish-services/windows/

[13] D. Fensel, F. van Harmelen, I. Horrocks, D. L. McGuinness, and P. F. Patel-Schneider, "OIL: an ontology infrastructure for the semantic web," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 38–45, Mar 2001.

[14] S. Groppe, D. Heinrich, and S. Werner, "Distributed join approaches for W3C-conform SPARQL endpoints," *Open Journal of Semantic Web (OJSW)*, vol. 2, no. 1, pp. 30–52, 2015. [Online]. Available: http://www.ronpub.com/publications/OJSW_2015v2i1n04_Groppe.pdf

[15] 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

[16] I. Horrocks, *The FaCT System*, 2003, accessed on 14.5.2016. [Online]. Available: http://www.cs.man.ac.uk/~horrocks/FaCT/

[17] 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.

[18] 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.

[19] M. Koubarakis, K. Kyzirakos, M. Karpathiotakis, C. Nikolaou, M. Sioutis, G. Garbis, and K. Bereta, *Introduction in stRDF and stSPARQL*, 2012, accessed on 27.7.2016. [Online]. Available: http://www.strabon.di.uoa.gr/files/stSPARQL_tutorial.pdf

[20] K. Kyzirakos, M. Karpathiotakis, and M. Koubarakis, *Strabon: A Semantic Geospatial DBMS*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 295–311.

[21] M. Lutz and D. Kolas, "Rule-based discovery in spatial data infrastructure," *Transactions in GIS*, vol. 11, no. 3, pp. 317–336, 2007.

[22] 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

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

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

[25] Open Geospatial Consortium, *OGC Catalogue Services 3.0 – General Model*, 2016, accessed on 4.8.2016. [Online]. Available: http://www.opengeospatial.org/standards/cat

[26] R. Pichler and S. Skritek, "Containment and equivalence of well-designed SPARQL," in *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '14. New York, NY, USA: ACM, 2014, pp. 39–50.

[27] E. Sirin and B. Parsia, "SPARQL-DL: SPARQL query for OWL-DL," in *OWLED*, 2007.

[28] M. Šmíd and Z. Kouba, "OnGIS: Methods of searching in spatial data driven by ontologies," in *Digitální technologie v geoinformatice, kartografii a DPZ*. Faculty of Civil Engineering, Czech Technical University in Prague, 2012, pp. 107–116.

[29] M. Šmíd and Z. Kouba, "OnGIS: Ontology driven geospatial search and integration," in *Terra Cognita 2012 Workshop*, ser. CEUR Workshop Proceedings, vol. 901. CEUR-WS.org, 2012, pp. 27–38.

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

[31] C. Stadler, J. Lehmann, K. Höffner, and S. Auer, "LinkedGeoData: A core for a web of spatial open data," *Semantic Web Journal*, vol. 3, no. 4, pp. 333–354, 2012.

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

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

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

[35] 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.

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

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

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

[39] W3C, *Linked Data*, 2015, accessed on 4.8.2016. [Online]. Available: https://www.w3.org/standards/semanticweb/data

[40] M. Wessel and R. Möller, "Flexible software architectures for ontology-based information systems," *Journal of Applied Logic – Special Issue on Empirically Successful Computerized Reasoning*, vol. 7, no. 1, pp. 75 – 99, 2009.

[41] World Wide Web Consortium, *OWL 2 Web Ontology Language: Profiles (Second Edition), OWL 2 EL*, 2012. [Online]. Available: https://www.w3.org/TR/owl2-profiles/#OWL_2_EL

[42] World Wide Web Consortium, *OWL 2 Web Ontology Language: Profiles (Second Edition), OWL 2 QL*, 2012. [Online]. Available: http://www.w3.org/TR/owl2-profiles/#OWL_2_QL

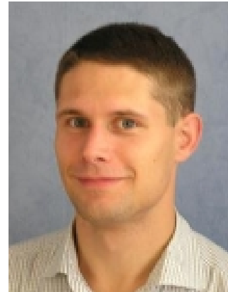
[43] C. Zhang, T. Zhao, and W. Li, "The framework of a geospatial semantic web-based spatial decision support system for digital earth," *Int. J. Digital Earth*, vol. 3, no. 2, pp. 111–134, 2010.

[44] 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.

[45] T. Zhao, C. Zhang, M. Wei, and Z.-R. Peng, "Ontology-based geospatial data query and integration," in *GIScience*, 2008, pp. 370–392.

AUTHOR BIOGRAPHIES

Marek Šmíd is a Ph.D. student in the field of artificial 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 artificial 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.