International Journal of Geo-Information

MDPI

Article

Adaptive and Optimized RDF Query Interface for Distributed WFS Data

Tian Zhao [1,*], Chuanrong Zhang [2] and Weidong Li [2]

[1] Department of Computer Science, University of Wisconsin-Milwaukee, Milwaukee, WI 53211, USA
[2] Department of Geography, University of Connecticut, Storrs, CT 06269, USA; chuanrong.zhang@uconn.edu (C.Z.); weidong.li@uconn.edu (W.L.)
* Correspondence: tzhao@uwm.edu

Academic Editors: Marinos Kavouras and Wolfgang Kainz
Received: 17 January 2017; Accepted: 31 March 2017; Published: 5 April 2017

Abstract: Web Feature Service (WFS) is a protocol for accessing geospatial data stores such as databases and Shapefiles over the Web. However, WFS does not provide direct access to data distributed in multiple servers. In addition, WFS features extracted from their original sources are not convenient for user access due to the lack of connection to high-level concepts. Users are facing the choices of either querying each WFS server first and then integrating the results, or converting the data from all WFS servers to a more expressive format such as RDF (Resource Description Framework) and then querying the integrated data. The first choice requires additional programming while the second choice is not practical for large or frequently updated datasets. The new contribution of this paper is that we propose a novel adaptive and optimized RDF query interface to overcome the aforementioned limitation. Specifically, in this paper, we propose a novel algorithm to query and synthesize distributed WFS data through an RDF query interface, where users can specify data requests to multiple WFS servers using a single RDF query. Users can also define a simple configuration to associate WFS feature types, attributes, and values with RDF classes, properties, and values so that user queries can be written using a more uniform and informative vocabulary. The algorithm translates each RDF query written in SPARQL-like syntax to multiple WFS GetFeature requests, and then converts and integrates the multiple WFS results to get the answers to the original query. The generated GetFeature requests are sent asynchronously and simultaneously to WFS servers to take advantage of the server parallelism. The results of each GetFeature request are cached to improve query response time for subsequent queries that involve one or more of the cached requests. A JavaScript-based prototype is implemented and experimental results show that the query response time can be greatly reduced through fine-grained caching.

Keywords: WFS; RDF; SPARQL; JavaScript; caching

1. Introduction

Web Feature Service (WFS) [1,2] is a Web-based protocol that supports feature-level access to geospatial data stores such as spatial databases, Shapefiles, and GML (Geography Markup Language) files. WFS services have been implemented in systems such as GeoServer [3] and MapServer [4]. Web interfaces such as those based on OpenLayers library [5] have been used to retrieve geospatial objects remotely over HTTP protocols and to display the results on a browser. This capability makes WFS services useful for a number of geospatial applications such as those for disaster management.

For example, New Haven, a coastal city in Connecticut, USA, has endured many storm disasters in the past. The August Tropical Storm Irene and the October Nor'easter in 2011 exposed the deficiency in geospatial data sharing in the response and management of the storm disasters. Under these situations, rapid prototyping of Web-based applications is necessary to allow disaster responders

and managers to share and update timely and accurate geospatial data over the network. To achieve this goal, a programming model is needed to allow programmers to rapidly collect various spatial data from multiple data sources for specific areas and to retrieve geospatial information for disaster response. However, directly using WFS for this purpose has two drawbacks.

1. WFS is designed for querying data in a single WFS server. To query data distributed in multiple WFS servers, a data client needs to be implemented. In particular, the user query may require spatial joins of data from different servers. When this happens, the data client has to send out multiple WFS GetFeature requests derived from the user query and then integrate the multiple WFS responses to produce an answer to the original query. It is more convenient from users' perspectives to consider the distributed data as centralized data so that a single user query can be used regardless of where the source data is located.

   In addition, user queries often involve the same intermediate results. Repeatedly requesting the same data adds unnecessary workload to the WFS servers, wastes network bandwidth, and increases the response time of the data client. This is especially critical under emergency situations when peak data requests might overwhelm data servers. Therefore, caching the results of the GetFeature requests is an important aspect for the implementation of the data client.

   For example, to display the flooded streets near high schools during a hurricane, where the school and street datasets are located in two WFS servers, it is necessary to first query each server with a WFS request and then integrate the extracted data from each server to get the answers. To handle these types of tasks, there are several possible design choices.

   The first choice is to provide a fixed set of user queries with predefined translation to WFS requests. However, users are limited to this set of queries and any new queries would require developers to revise the data client, which is not suitable for emergency situations.

   The second choice is to provide a separate query interface for each data source and allow users to overlay the query results on a base map to achieve the visual effect of spatial joins. However, if the different datasets have semantic heterogeneity problems, it is difficult to do this without human assistance or conversion. For example, a WFS may advertise as a "freeway"-feature data provider, whereas another WFS may advertise as a "highway"-feature data provider even though the "freeway" feature and the "highway" feature refer to the same concept. Users cannot automatically compose and synthesize the two feature datasets together without additional human interpretation or conversion.

   The third choice is to first integrate the distributed data into a flexible format such as Resource Description Framework (RDF) and then query the integrated data through an RDF query engine. However, this approach is not scalable to large datasets. In addition, for frequently updated geospatial data such as that for emergency management, it is infeasible to convert WFS features into RDF format each time there is a change.

2. The definitions of WFS features are often auto-generated from the underlying data sources, which may not correspond to the definitions of the same concepts in user applications.

   For example, to retrieve high schools near state highways and display the results on a map, a user has to find out the feature types, feature attributes, and values for encoding the suitable WFS requests. For example, the state highways and schools are contained in the feature type `new_haven_road` and `new_haven_places`, respectively, and the filters are

$$\begin{array}{c} \text{RTTYP } \textit{PropertyIsEqualTo } \text{“S”} \\ \text{CATEGORY } \textit{PropertyIsEqualTo } \text{“High School”}, \end{array} \tag{1}$$

   where `RTTYP` is the attribute in `new_haven_road` to identify road type and `S` is the value for state highway while `CATEGORY` is the attribute in `new_haven_places` to identify the type of the place.

Users have to guess the meaning of the feature types and attributes each time they want to encode a GetFeature request. Simple mistakes such as typographical errors can cause a GetFeature request to fail. Moreover, similar concepts at user level may be stored by different WFS features. For instance, to find the local roads, a user needs to query the feature type `new_haven_street` with the filter `LEGEND PropertyIsEqualTo "Local Road"`.

The new contribution of this study is to propose a novel adaptive and optimized RDF query interface for distributed WFS to overcome the aforementioned limitations using WFS. The proposed data query interface can avoid the drawbacks of the aforementioned choices by automatically translating each user query to WFS requests, performing spatial joins of the WFS results if necessary, and caching the intermediate results to reduce server workload, network traffic, and data client response time. In addition, the proposed query interface can facilitate the rapid prototyping of WFS query applications. The RDF query interface translates a user query formulated in a SPARQL-like [6] syntax to WFS requests sent to multiple servers and then integrates the multiple WFS responses to answer the original query. Using this design, programmers do not need to write code to communicate with WFS servers, transform their responses, or integrate the results. Instead, they can accomplish the same goal by defining mappings from WFS feature types to RDF definitions and making RDF queries. Simple mapping rules to associate WFS feature types, attributes, and values to RDF classes are designed so that user queries can be written in higher level concepts. The algorithm translates each RDF query written in SPARQL-like syntax to multiple WFS GetFeature requests, and then converts and integrates the multiple WFS results to get the answers to the original query. The generated GetFeature requests are sent asynchronously and simultaneously to WFS servers to take advantage of the server parallelism. The results of each GetFeature request are cached to improve query response time for subsequent queries that involve one or more of the cached requests. A JavaScript-based prototype is implemented and experimental results show that the query response time can be greatly reduced through fine-grained caching.

For the rest of the paper, we first give an overview of the proposed query interface in Section 2, and then we review the related works in Section 3. We provide details for mapping RDF definitions to WFS feature types in Section 4 and details for implementing our RDF query interface are provided in Section 5. The implementation and experimental results are explained in Sections 6 and 7. We discuss possible future work in Section 8 and conclude in Section 9.

2. Overview

In this section, we give an overview of the proposed RDF query interface using an example and explain the primary benefits to users.

2.1. RDF Query Interface

The data provided by WFS servers may be originated from many sources. We choose RDF to define a uniform data model due to its wide acceptance and its support for data integration and interoperability at the semantic level for the Semantic Web. Other types of data definitions may work as well. For example, Query (2) is an RDF query for selecting the state highways near high schools, where a geometry is *close to* another one if their distance is less than 500 m (this distance is arbitrarily chosen and can easily be modified):

```
select ?r, ?p   where
                ?r rdf:type roads.
                ?r nh:type "State Highway".
                ?p rdf:type places.
                ?p nh:category "High School".
                ?r nearby ?p.
```
(2)

In the query, identifiers start with *?* represent variables. The solutions to the variables *?r* and *?p* (which stand for roads and points, respectively) between *select* and *where* are the intended results of the query. The lines after *where* are called triples that specify the conditions with which the variable solutions must satisfy. Each triple has the form of *subject predicate object*, which restricts the relationship between the subject and object with the predicate. For example, the triple *?r rdf:type roads* says that the solution to *?r* must has a type called *roads*. The triple *?p nh:category "High School"* specifies that the category of *?p* is *High School*. The namespace prefix of predicate *nh:category* is *nh*, which stands for New Haven. Finally, the triple *?r close to ?p* relates the spatial attributes of *?r* to those of *?p* by the distances between them.

The RDF definitions in this example are related to the WFS feature types through a set of mapping rules that associate WFS feature types, attributes, and values to RDF classes, properties, and values, respectively. Through this mapping, feature types and attributes can be transformed into more standard RDF classes and properties with semantic meanings. Additional RDF classes and properties can be defined from the ones directly mapped from feature types. This would allow more succinct representation of RDF queries. For example, we can redefine Query (2) as below:

```
select ?r, ?p   where
                ?r rdf:type state_highway.
                ?p rdf:type high_school.
                ?r nearby ?p,
```
(3)

where the class `state_highway` and `high_school` are derived from the RDF classes `roads` and `places` mapped from the feature types `new_haven_road` and `new_haven_places`. The additional definitions of RDF classes allow us to write RDF queries that are more flexible than the equivalent WFS queries.

Note that the query language accepted by our RDF interface is not standard SPARQL or GeoSPARQL [7], which is an extension of SPARQL with spatial functions and predicates. Our query language is limited to triple statements and does not have a filtering component. The purpose of our RDF query interface is to facilitate data retrieval from distributed WFS data so that RDF queries can be automatically translated to WFS queries. This translation may not be possible if a user query is based on the full syntax of SPARQL or GeoSPARQL.

2.2. RDF Queries as WFS Requests

The RDF query interface proposed in this paper converts a user query to one or more WFS GetFeature requests, sends each request to one of the WFS servers, and then integrates the queried results from each WFS server. The interface allows users to write queries using high-level concepts defined through a set of mapping rules that associate RDF definitions with low-level WFS feature types. The interface also handles sending asynchronous queries to WFS servers, transforming, caching, and integrating the query results with spatial joins if necessary.

Compared to the approach of sending WFS requests separately and integrating the results, our RDF query interface reduces the workload of coding WFS requests for each application and makes it possible to improve performance by caching the intermediate results of user query. Thus, the spatial data stored in WFS servers can be independently updated and efficiently queried. Spatial relations can be defined as virtual RDF properties that are not materialized ahead of time. Instead, they are computed on demand, which is more efficient for cases where a small number of spatial objects are involved.

3. Related Work

Geospatial data can be queried directly from a relational database such as Oracle database, MySQL, and PostgreSQL with spatial extensions. The typical query language used is SQL with spatial filters. Spatial indexing can be used to speed up querying of spatial objects such as points and lines. However,

accessing a database directly has its drawbacks as different databases have different connections and query mechanisms. To this end, standard OGC (Open Geospatial Consortium) protocols such as WFS and WMS were developed to enable access to spatial data stored in traditional sources such as spatial databases and proprietary formats such as Shapefiles through a uniform format. Web servers such as GeoServer that implement WFS and WMS provide convenient access to data within a server through a browser client such as OpenLayers. GeoServer also supports spatial indexing. While WMS protocol provides access to spatial data as map images, the WFS protocol provides access to spatial data as features where each feature represents spatial objects such as a point, a line, or an area feature that also have a number of attributes such as names, length, and category. The primary mechanism for retrieving data from a WFS server is to use the GetFeature request, where users can specify the filtering criteria. However, querying distributed WFS data requires coordination at the client side where users of the map client may have to perform multiple queries and manually integrate the results.

Substantial work has been done in the context of Semantic web services, such as the efforts on OWL for Services (OWL-S) [8], the Web Services Modeling Ontology (WSMO) [9,10], and WSDL-S [11]. In literature, declarative logic programs using Prolog have also been used for knowledge representation purposes and allow RDF and SPARQL queries [12–14]. In the geospatial community, in order to facilitate discovery of geospatial web services compliant with OGC specifications and promote access to geospatial information via the OGC geospatial web services, efforts have been made to incorporate ontology into OGC geospatial web services (e.g., [15–20]). There are recent interests by researchers in exploiting the geospatial semantic web for automatic integration of semantically heterogeneous geospatial data (e.g., [19,21–32]).

In literature, the connection between geospatial web services and ontology has been explored from several directions. Ontology has been used in search tools to help to discover geospatial web services related to certain domain concepts [33]. Tools have also been developed to convert geospatial ontology data to forms that can be accessed via WFS protocol [34]. Given the abundance of data available from geospatial web services and databases, a more interesting direction is to make data from geospatial web services and databases accessible via ontological query protocols.

Although a substantial amount of work has been done to study the access of non-spatial relational database data as ontological data (see [35] for detailed comparison of different approaches), to our best knowledge, there are only a few studies in literature to investigate access of WFS data via ontology query protocol.

The closest study is the work of Tschirner et al. [36], who proposed a method to convert GML data into ontology data by translating SPARQL queries to WFS requests. This approach is to map a SPARQL query to a WFS request that returns a superset of intended results, transform the WFS results into ontology data, and then apply the original SPARQL query to obtain the final answer. While this paper has a similar workflow, our approach is different in several ways. Firstly, we do not assume the WFS data is centralized in one server or has a unified definition. This requires the translated WFS requests to be separated for each feature type and the final join to be done at the client side. We generate multiple OGC filter encodings for each SPARQL query. Secondly, our approach implements a light-weight Web client with roughly 1000 lines of JavaScript without library dependencies for query processing, which is easier for deployment, while Tschirner et al. implemented a server program with dependencies such as Sesame and Deegree3 API (Application Program Interface). Lastly, our approach is intended to use a SPARQL-like syntax to bring more convenient query interface to WFS services, while Tschirner et al. provided a more complete service of SPARQL endpoints for GML data sources.

In our prior work [31], we proposed a query rewriting algorithm to translate SPARQL query to WFS requests and database queries using idealized syntax. This paper extends that approach by considering a more realistic subset of SPARQL syntax and by implementing a Web-based prototype that incorporates caching optimization and data rendering.

Li et al. [33] developed a prototype to chain geospatial web services for accessing arctic geospatial data. Their basic idea is to use the ontology query engine to search for concepts that are linked to

features available from the known OGC geospatial web services and then query the located web services. However, how to make connections between spatial ontology and geospatial web services is not the focus of their study. In their study, only some concepts in the ontology have been mapped to the feature types in geospatial web services. The located geospatial web services are queried via web service protocol (e.g., WMS), which may not conform to the ontology representation. The features in the linked geospatial web services cannot be directly queried as ontology instances using protocols such as SPARQL or GeoSPARQL.

More recently, Jones et al. [34] developed a tool, LOD4WFS [37], to convert spatial ontology data to a form that can be used by WFS clients. Their tool first uses predefined SPARQL queries to select spatial ontology data, and then converts the selected spatial ontology data to WKT format that can be returned as the result of WFS queries. Their tool handles three WFS services requests: GetFeature, DescribeFeatureType, and GetCapabilities. The GetFeature request seems to be limited in that it can only select all feature instances and their properties without filtering. Performance comparison results in that study showed that the runtime performance is similar to that of the traditional backend of WFS services when requesting all feature instances. While this tool allows the spatial ontology data to be used by geospatial web service clients, its utility is not clear since the filtering capability of WFS, which is important for many applications, is not supported. In addition, the flexibility of ontology query is lost by using the predefined SPARQL queries to handle WFS requests.

4. RDF Mapping

In order to treat WFS features as RDF data, we first use simple relational rules to map feature definitions to RDF definitions. Specifically, for the motivational example, several WFS feature types for New Haven, Connecticut and its surrounding areas are used. The feature types include streets, roads, and places. Both street and road features are line geometries. Street are generally shorter than the roads but streets have more detailed attributes than roads. The places are point geometries for locations such as schools, bridges, and railroad stations.

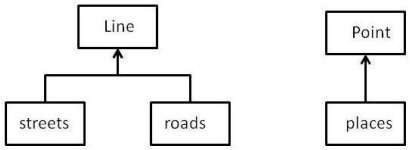
For our prototype implementation of the proposed RDF interface, a few RDF classes are created for the motivational example. As shown in Figure 1a, the RDF classes *streets*, *roads*, and *places* are mapped to the WFS feature types `new_haven_streets`, `new_haven_roads`, and `new_haven_places`, respectively. Figure 1b illustrates the RDF classes created for the prototype. The RDF class *Line* is the parent RDF class of *streets* and *roads*, while the RDF class *Point* is the parent RDF class of *places*. Additional classes can be introduced to the hierarchy without making any change to the implementation. For instance, this example does not use polygon geometry though we can support it by (1) defining a base class for polygon, (2) adding more RDF classes that extend the polygon class, and (3) defining additional rules to associate these RDF classes to WFS feature types that have polygon geometries.

The RDF properties are mapped to the feature type attributes. For example, Figure 1c is the mapping between the properties of the *roads* and *streets* and the attributes of `new_haven_road` and `new_have_street`, respectively. Similar mappings are defined for values of some attributes to provide more readable query results. For example, Figure 1d is the mapping between the values of the attribute *RTTYP* [38] to the values of the property *nh:type* in the roads class. Mapping feature definitions to RDF definitions clarifies the vocabulary of the original data. Cryptic names such as *RTTYP* become self-evident names such as *nh:type*. Abbreviations such as the values *S* and *I* become full terms such as *State Highway* and *Interstate*. In addition, attributes *RTTYP* and *LEGEND* have similar meanings so that they are mapped to the same property *nh:type* (not *rdf:type*).

(**a**)

feature types	RDF classes
`new_haven_streets`	streets
`new_haven_roads`	roads
`new_haven_places`	places

(**b**)



(**c**)

Attributes (streets)	Properties
FENAME	nh:name
FETYPE	nh:suffix
LEGEND	nh:type

Attributes (roads)	Properties
FULLNAME	nh:name
RTTYP	nh:type

(**d**)

Values of *RTTYP*	Values of *type*
C	County
I	Interstate
M	Common name
O	Other
S	State Highway
U	US

(**e**)

RDF class	Definition in SPARQL triples
high_school(?x)	?x rdf:type places. ?x nh:category “High School”.
state_highway(?x)	?x rdf:type roads. ?x nh:type “State Highway”.

Figure 1. (**a**) mapping of feature types, (**b**) ontology class hierarchy, (**c**) mapping of attributes of roads and streets, (**d**) mapping of the values of *RTTYP*, (**e**) definition of RDF class high_school and state_highway.

Figure 1e illustrates the way that additional RDF classes are defined based on the ones directly mapped from feature types. For example, we define the class `high_school` by mapping it to SPARQL triples using a function

```
high_school(?x) = ?x rdf:type places.
                  ?x nh:category "High School".
```

Given this definition, we expand any triple of the form `?y rdf:type high_school` in an RDF query to `?y rdf:type places. ?y nh:category "High School"`. Note that the variable `?x` is replaced by the variable `?y` in the translation. After query expansion, the query in Label (3) is translated to the query in Label (2). The expanded query is then translated to two GetFeature requests to retrieve features for roads and places with the filters in Label (1).

5. Translate RDF Query to WFS Requests

In the following subsections, we first explain the syntax of the SPARQL queries that we support and the syntax of corresponding WFS requests, and then we explain the query translation algorithm.

5.1. Syntax

We consider a simplified form of SPARQL queries with the syntax in Figure 2. We choose this subset of SPARQL syntax since it approximates the capabilities of WFS GetFeature requests. It is sufficient for expressing the common queries such as selecting features based on constraints between features and constraints on feature attributes. The full SPARQL protocol contains additional structures such as union and optional triple patterns that can be simulated with multiple SPARQL queries and post-query processing. However, supporting these capabilities is not essential to our goal of accessing distributed WFS data.

Q	::=	`select` v `where` P	query
v	::=	$?x \mid ?x,\ v$	variables
P	::=	$s\ p\ o$	triple pattern
	$\mid$	$P.P'$	conjunction
$?x$	$\in$	V	variable
s	$\in$	$U \cup V$	subject
p	$\in$	U	predicate
o	$\in$	$U \cup V \cup L$	object

R	::=	`GetFeature`(t, f)	WFS request
f	::=	$pp\ op\ d$	filter
	$\mid$	$f \wedge f'$	conjunction
t	$\in$	$rdf{:}type$	feature type
pp	$\in$	`Prop`	feature property
op	$\in$	`Op`	operator
d	$\in$	`Value`	property value

U : the set of URIs V : the set of variables L : the set of literals

Figure 2. Syntax of SPARQL queries and the corresponding WFS requests.

Figure 2 (left) contains the BNF grammar (BNF grammar is a common notation for context-free grammars, which are often used for representing the syntax of programming languages.) for our SPARQL queries, where a query Q of the form *select* v *where* P consists of a set of selection variables v and a triple pattern P. A triple pattern is a conjunction of a set of triples separated by dots. For example, the conjunction of P and P' is represented as $P.P'$. A triple of the form $s\ p\ o$ consists of a subject s, a predicate p, and an object o. A subject is either a URI (Uniform Resource Identifier) or a variable, a predicate is a URI, and an object can be a URI, a variable, or a literal. A URI includes a prefix and a short-name to identify an RDF resource. Note that in full SPARQL the predicate can also be a variable. In this paper, we only consider the URI that refers to an RDF class or a property. The answer to the query is a set of tuples where each tuple is a solution to the variables. We restrict the selection variables v to be the ones that correspond to geospatial features such as lines or points.

In order to provide a succinct description of the algorithm for translating SPARQL queries, we also define a formal syntax for the WFS GetFeature requests generated from SPARQL queries. Figure 2 (right side) contains a BNF grammar for the GetFeature requests, each of which consists of a feature type t and a set of filters f on feature properties. GetFeature requests can limit the set of returned feature properties, but all feature properties will be returned by default. For simplicity, the grammar only considers GetFeature requests that return all feature properties. A filter of the form $pp\ op\ d$ allows the query to select features so that $op(pp, d)$ is true, where op is an operator function, pp is a property of the features, and d is a data value that the feature property must be related to through the operator.

5.2. Translation Algorithm

Typically, a SPARQL query will be translated to multiple WFS requests. To reduce network traffic, our algorithm reduces the number of WFS requests by grouping triples related to the same feature type together so that there is one WFS request per feature type. Any remaining triples are translated to spatial joins to be applied to the results of the WFS requests. Figure 3 is an overview of the translation algorithm, which includes a pre-processing, a rewriting, and a post-processing phase.

5.2.1. Pre-Processing

The pre-processing phase separates the triples into four groups based on the triple predicates: type triple, where the predicate is *rdf:type* that specifies the RDF class of the triple subject; property triple, where the predicate corresponds to a geospatial feature property; filter triple, where the predicate is a comparison operator that compares the subject with a literal; and spatial join triple, where the predicate is a spatial operator that joins two geospatial features. For example, the triples in Query (2) can be separated as in Table 1.

Table 1. Separated triples of Query (2).

type triple	*?r rdf:type roads.* *?p rdf:type places.*
property triple	*?p nh:type ?t.* *?p nh:category ?c.*
filter triple	*?t == "State Highway".* *?c == "High School".*
spatial join triple	*?r nearby ?p*

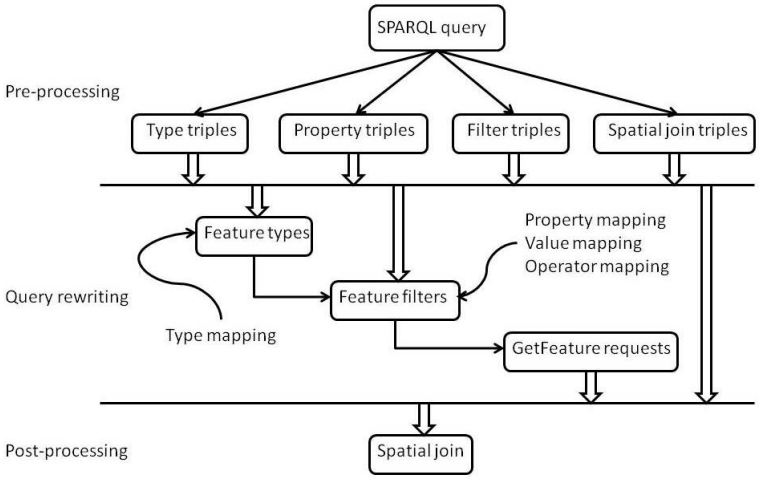


Figure 3. Overview of the query translation algorithm.

Note that the triples *?p nh:category ?c* and *?c == "High School"* are automatically generated from the one triple *?p nh:category "High School"*, where the variable *?c* is a fresh.

5.2.2. Query Rewriting

Based on the separated triples, the rewriting phase includes five steps. The first step is to identify the set of feature variables fv that correspond to feature types. The second step is to find the set of triples related to each feature variable. The third step is to find the set of feature types for each feature variable. The fourth step is to find the set of filters for each pair of feature variable and its type. The fifth step is to construct a set of WFS GetFeature requests for each feature variable.

We explain the five steps in detail as follows. In the first step, we find the query variables fv that correspond to feature types:

$$\begin{aligned} fv &= \{?x \mid (?x\ p\ o) \wedge \texttt{isProp}(p)\} \\ &\cup \{?x \mid ?x\ rdf{:}type\ o\} \\ &\cup \{?x, ?y \mid (?x\ p\ ?y) \wedge \texttt{isSpatial}(p)\}. \end{aligned}$$

The identification of fv is based on the above equation. If there exists a triple *?x p o*, where the predicate p corresponds to a geospatial feature property ($\texttt{isProp}(p) = true$) or there is a triple *?x rdf:type o*, then the subject $?x$ of the triple is considered a query variable. In addition, if there exists a triple *?x p ?y*, where p corresponds to spatial join operator ($\texttt{isSpatial}(p) = true$), then the subject $?x$ and the object $?y$ of the triple are considered query variables. For Query (2), we can identify feature variables $?r$ and $?p$ through the triples *?r rdf:type streets*, *?p nh:category ?c*, and *?r nearby ?p*.

In the second step, we find the set of type triples, property triples, and filter triples related to each variable in fv using the below equations, where the function $\texttt{isOp}(p)$ returns true if p corresponds to a WFS filter operator. The three sets of triples are the building blocks for generating the WFS GetFeature requests:

$$\begin{aligned}
\texttt{typeTriple}(?x) &= \{?x\ rdf{:}type\ o\} \\
\texttt{propTriple}(?x) &= \{?x\ p\ ?y \mid \texttt{isProp}(p)\} \\
\texttt{filterTriple}(?x) &= \{?y\ p\ o \mid \texttt{isOp}(p) \\
&\quad \wedge \quad ?x\ p'\ ?y \in \texttt{propTriple}(?x)\}.
\end{aligned}$$

In the third step, we compute the set of feature types $ft(?x)$ to which a variable $?x$ may resolve. We first define a function ft_1 to compute a set of types to which $?x$ corresponds using the triples that restrict the class of $?x$, where the function $subclass(o)$ returns the subclasses of o, which is an RDF class such as points, lines, roads, streets, or places, and the function $\texttt{typeMapping}(c)$ returns the feature type mapped to c. In other words, if a triple says that $?x$ must be of type o, then the type of $?x$ must be a subclass of o:

$$\begin{aligned}
ft_1(?x) &= \{\texttt{typeMapping}(c) \mid c = subclass(o) \\
&\quad \wedge \quad ?x\ rdf{:}type\ o \in \texttt{typeTriple}(?x)\}.
\end{aligned}$$

For example, $ft_1(?r) = \{\texttt{new\_haven\_roads}\}$ because $?r\ rdf:type\ roads \in \texttt{typeTriple}(?r)$ and the only subclass of *roads* is itself and its corresponding feature type is `new_haven_roads`. However, if there exists a triple *?r rdf:type Line*, then $ft_1(?r) = \{\texttt{new\_haven\_streets}, \texttt{new\_haven\_roads}\}$ since both streets and roads are subclasses of *Line*.

We also define a function ft_2 to compute the set of types that $?x$ corresponds to the triples that use property predicates, where the function $\texttt{domain}(p)$ returns the domain class of the predicate p. That is, for each triple of the form *?x p ?y*, if p is a property predicate, then the type of $?x$ must be in the domain of p:

$$\begin{aligned}
ft_2(?x) &= \{\texttt{typeMapping}(c) \mid c = \texttt{domain}(p) \\
&\quad \wedge \quad ?x\ p\ ?y \in \texttt{propTriple}(?x)\}.
\end{aligned}$$

For example, from *?p nh:category ?c*, we have $ft_2(?p) = \{\texttt{new\_haven\_places}\}$, since the domain of *nh:category* is *Places*, which corresponds to the feature type `new_haven_places`. Finally, the set of feature types that $?x$ corresponds to is the intersection of $ft_1(?x)$ and $ft_2(?x)$ if both of them are not empty. If one of them is empty, it is equal to the non-empty set. If both are empty, then it is equal to Type, which is the entire set of WFS feature types:

$$ft(?x) = \begin{cases} ft_1(?x) \cap ft_2(?x) & \text{if } ft_1(?x) \neq \emptyset \wedge ft_2(?x) \neq \emptyset \\ ft_1(?x) & \text{if } ft_1(?x) \neq \emptyset \wedge ft_2(?x) = \emptyset \\ ft_2(?x) & \text{if } ft_1(?x) = \emptyset \wedge ft_2(?x) \neq \emptyset \\ Type & \text{otherwise} \end{cases}.$$

The last case applies to any feature variable that is not restricted in any way except that it appears in a spatial triple. For example, in Query (4), since $?r$ is not restricted, $ft_1(?r) = \emptyset \wedge ft_2(?r) = \emptyset$ and $ft(?r) = \{\texttt{new\_haven\_places}, \texttt{new\_haven\_streets}, \texttt{new\_haven\_roads}\}$:

```
select ?r, ?p   where
                ?p rdf:type high_school.
                ?r nearby ?p.
```
(4)

In the fourth step, we find the set of filters using the function $Filter(t, ?x)$ for each feature type t in $ft(?x)$, where $\texttt{propMapping}(t, p)$ returns the property pp of feature type t corresponding to p,

$\texttt{valueMapping}(t, o)$ returns the value d of feature type t corresponding to o, and $\texttt{opMapping}(p)$ returns the WFS operator that corresponds to p. Each filter has the form of $pp\ op\ d$, which says that the property pp of t must satisfy the Boolean formula $op(pp, d)$:

$$\begin{aligned}\texttt{filter}(t, ?x) \quad = \quad & \{pp\ op\ d \mid ?x\ p'\ ?y \in \texttt{propTriple}(?x) \\ \wedge \quad & ?y\ p\ o \in \texttt{filterTriple}(?x) \\ \wedge \quad & op = \texttt{opMapping}(p) \\ \wedge \quad & pp = \texttt{propMapping}(t, p') \\ \wedge \quad & d = \texttt{valueMapping}(t, o)\}.\end{aligned}$$

For example, for Query (2), the feature variables are ?r and ?p and their feature types are `new_haven_roads` and `new_haven_places`, respectively. The following filters can be derived:

$$\begin{aligned}\texttt{filter}(\textit{new\_haven\_roads}, ?r) \quad &= \quad \{\text{RTTYP } \textit{PropertyIsEqualTo } \text{`S'}\} \\ \texttt{filter}(\textit{new\_haven\_places}, ?p) \quad &= \quad \{\text{CATEGORY } \textit{PropertyIsEqualTo } \text{`High School'}\}\end{aligned}.$$

Lastly, the function $\texttt{gf}(?x)$ returns a set of GetFeature requests for each variable $?x$ in fv, where f is a conjunction of filters in $\texttt{filter}(t, ?x)$. Each of the GetFeature requests consists of a feature type t that corresponds to the variable $?x$ and the conjunction of the filter f_i related to t:

$$\begin{aligned}\texttt{gf}(?x) \quad = \quad & \{\texttt{GetFeature}(t, f) \mid f = f_1 \wedge f_2 \wedge \ldots \wedge f_n \\ \wedge \quad & (\forall i. f_i \in \texttt{filter}(t, ?x)) \\ \wedge \quad & t \in ft(?x)\}.\end{aligned}$$

Using this function, we can generate the following GetFeature requests from the previous query:

$$\begin{aligned}&\texttt{GetFeature}(\textit{new\_haven\_roads}, \text{ RTTYP } \textit{PropertyIsEqualTo } \text{`S'}) \\ &\texttt{GetFeature}(\textit{new\_haven\_places}, \text{ CATEGORY } \textit{PropertyIsEqualTo } \text{`High School'}).\end{aligned}$$

5.2.3. Post-Processing

After receiving the responses from the GetFeature requests, we proceed to the post-processing phase by performing spatial joins on the retrieved geospatial features if necessary. The results of the spatial join operation are filtered by the types of the features identified from the selection variables of the SPARQL query. Note that in order to support data retrieval from multiple WFS servers, we retrieve each type of features individually and then apply the spatial join operation at the client side.

In this phase, we define the function $\texttt{feature}(?x)$ to execute the GetFeature requests of $?x$ and run the spatial joins on the results, where the call $\texttt{result}(\texttt{gf}(?y))$ obtains the features of the GetFeature requests $\texttt{gf}(?y)$. The features corresponding to the query variable $?x$ are a subset of the results from the GetFeature requests $\texttt{gf}(?x)$, so that each feature F of the subset must satisfy the spatial join triple $F\ p\ F'$ for each triple $?x\ p\ ?y$ in the query, where p is a spatial join operator and F' is a feature returned from the GetFeature requests $\texttt{gf}(?y)$:

$$\begin{aligned}\texttt{feature}(?x) \quad = \quad & \{F \in \texttt{result}(\texttt{gf}(?x)) \mid \forall ?x\ p\ ?y.\ \texttt{isSpatial}(p) \\ \wedge \quad & F\ p\ F' \\ \wedge \quad & F' \in \texttt{result}(\texttt{gf}(?y))\}.\end{aligned}$$

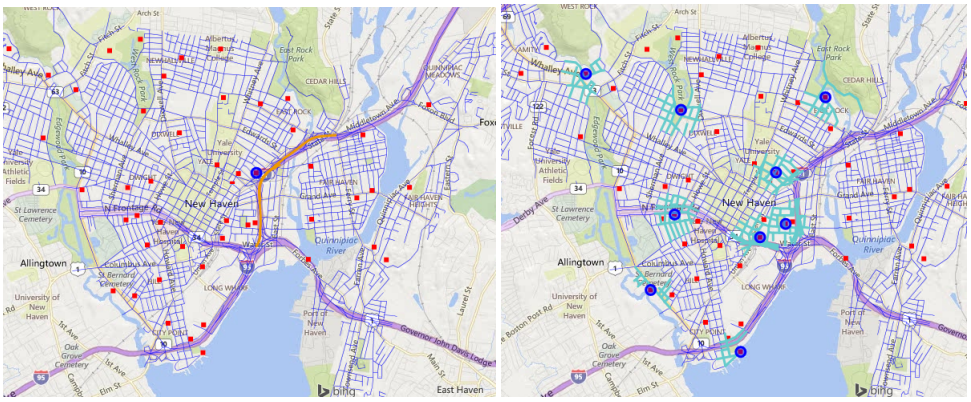
Finally, given a query *select v where P*, we obtain $\{\texttt{feature}(?x) \mid ?x \in v\}$. Though the previous steps imply an order in which features are retrieved, in actual implementation, the features retrieved from the GetFeature requests $\texttt{gf}(?x)$ for each $?x \in v$ are cached. We perform the spatial join for each spatial join triple $?x\ p\ ?y$ using the retrieved features as inputs. This caching step can dramatically

improve performance for multiple queries since the data retrieved from the previous queries may be reused in the subsequent queries to avoid resending the same GetFeature requests to remote servers.

For example, Query (2) is translated to two GetFeature requests with one retrieving state highway roads and the other retrieving high schools. The two sets of features are spatially joined based on distance.

6. Implementation

The proposed algorithm is implemented in JavaScript for the data sets of our running example. It parses each SPARQL query in text form and generates a set of GetFeature requests, which are sent as AJAX calls to WFS servers and the responses are joined before being displayed on a map. For example, Figure 4a shows middle schools near interstate roads, where the blue-colored circle represents the only middle school identified and the orange-colored lines represent the nearby interstate roads. Figure 4b shows streets near high schools, where the cyan-colored lines represent streets near high schools (blue-colored circles). The implemented software is available at [39], where users can display a map found on the left and a text box on the right.



(a) (b)

Figure 4. (**a**) Interstate near middle school and (**b**) streets near high schools.

There are some pre-defined queries users can select from the drop-down menu though they are not hard-wired in the implementation in any way and users can revise the queries in the textbox. Users can click the *query* button on the bottom right to start the query and the results will be displayed on the map. For example, Figure 5 shows the result of Query (2/3) that selects state highways near high schools. The buttons on the top row of the map give users the ability to select streets, roads, and places directly using the mouse. Users can also click the *block streets* or *block roads* button to mark the selected streets and roads as being blocked (possibly due to weather conditions). Once a street is marked as blocked, the map for the street layer will show it in a different color, which indicates that this change is saved in the WFS server. Note that the tested WFS server is transactional and our implementation supports persistency through direct calls to the WFS server, not through RDF query interface.

6.1. Feature Retrieval and Callbacks

The displayed map is implemented using OpenLayers, which is a JavaScript library for interactive maps that can connect to OGC web services such as WFS and WMS services. The map includes a base layer from the Bing map service and several of the WFS layers that are dynamically updated with features generated from our query translation algorithm. Since JavaScript is a single-threaded language, it uses an event-based model to handle asynchronous communication with remote servers.

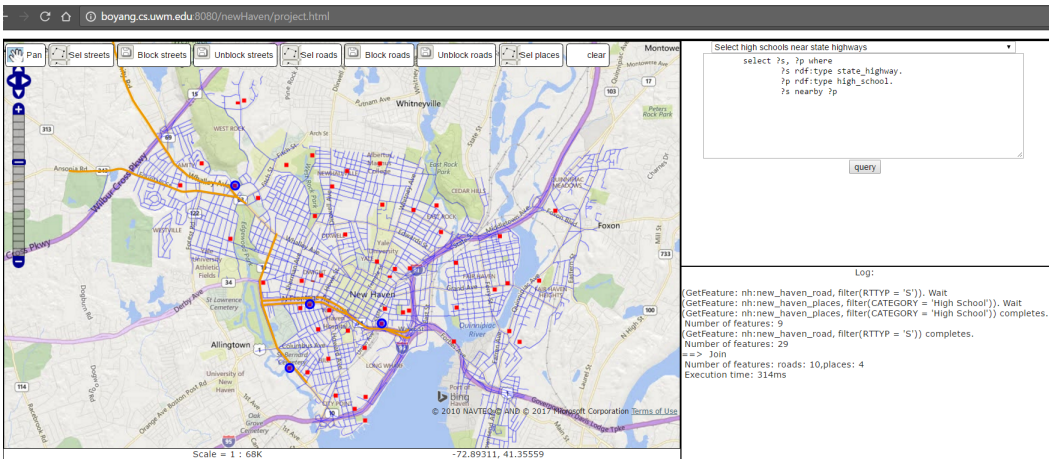


Figure 5. State highways near high schools.

For this implementation, AJAX calls are used to fetch features from WFS servers. However, there may be multiple WFS requests for each RDF query and the final results of these requests may need to be spatially joined. Since each AJAX call is implemented as a callback function that waits for the response event of a HTTP request, the AJAX call cannot return the features that it retrieved to the main thread. To use the retrieved features, we pass a callback function to each AJAX call that handles a GetFeature request so that the callback will only perform the spatial join operation after all AJAX calls have returned. Figure 6 illustrates how multiple GetFeature requests are implemented. The callbacks are triggered by the AJAX events, which will occur when responses to the GetFeature requests are received.

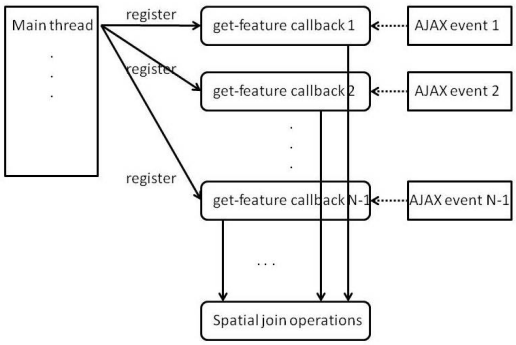


Figure 6. Callbacks.

6.2. Caching

Naive implementations of web service-based map applications may retrieve the same sets of features over the network repeatedly, which can cause significant performance problems not only for the map clients but also for the servers. The most common way of handling this problem is to cache the retrieved sets of features so that subsequent requests of the same features can use the cached results. However, directly caching the results of the RDF queries is too fine-grained for our algorithm since it is difficult to determine whether two RDF queries involve the same set of features. Instead, we cache the results of GetFeature requests translated from the previous RDF query. For the next RDF query, we compare the translated GetFeature requests with the cached ones and use the cached results if available. This caching optimization greatly improves the runtime performance and avoids placing unnecessary strain on the WFS servers. In our prototype implementation, users can see the query log, which indicates whether a WFS request is answered through a GetFeature request or from the cache.

7. Evaluation

To evaluate the performance of our interface, we have conducted a set of query experiments using a data set consisting of three WFS features, where the number of instances and the retrieval time of each feature are displayed in Table 2.

Table 2. The number of instances and the retrieval time for each feature type.

	Streets	**Roads**	**Places**
# of instances	3449	1537	54
query time	3910 ms	1200 ms	88 ms
total time	6632 ms	1832 ms	131 ms

The total time includes the query time (including the time for WFS server response and network transmission) and the time for rendering the WFS features on the map.

Our experiments tested the following list of queries, where queries 1–7 involve spatial joins of two features, query 8 involves spatial joins of three features, and queries 9–11 do not have spatial joins:

1. Select high schools near state highways;
2. Select roads near middle schools;
3. Select streets near high schools;
4. Select streets near "Fair Haven School" and less than 500 feet long;
5. Select highways near high schools;
6. Select the streets near the road "Connecticut Tpke";
7. Select the middle schools near interstates;
8. Select the elementary schools and streets near the road "Connecticut Tpke";
9. Select streets and roads of the name "Lawrence";
10. Select features of the name "I-95";
11. Select features of the name "Q Bridge".

The number of feature instances involved in each query is shown in Table 3 and the runtime performance of the queries is shown in Table 4. The runtime of each query is measured (1) without caching, (2) with caching of one feature that has the most number of instances, and (3) with caching of all features. Caching significantly reduces the runtime cost of all queries. The improvement is significant for queries that involve the retrieval of a large of instances from WFS servers such as queries 2–6 and 8. Note that even when all features are cached, the runtime of queries 6 and 8 is still over one second, which is significantly higher than the other queries. This difference is mostly due to the cost of spatial join between streets and roads (both polylines), which is much higher than the cost of spatial join between streets and places (polylines and points).

The last two columns of Table 4 show the size of the data returned from each WFS GetFeature request and the time for servers to generate responses to the GetFeature request and to transmit the responses to the browser client. The data size and GetRequest response time are measured using Chrome browser tools. For example, for Query 3, there are two GetFeature requests sent to the servers and the sizes of the GetFeature responses are 394 and 1.3 Kilo-bytes (kb) each, while the corresponding server/network time is 3900 and 97 milli-seconds (ms) each. The total response time of Query 3 is 578 ms. By caching one or both the responses of the GetFeature requests, we are able to reduce the total response time to 214 ms and 81 ms, respectively.

This experiment is conducted in a realistic setting where two WFS servers that host distributed WFS features are located in Milwaukee, WI, USA, while the client browser is located in Chicago, IL, USA. Each of the GetFeature requests is sent separately from the client and answered by the servers in parallel.

In summary, through caching, we are able to achieve speed-up of query response time (Figure 7) ranging from 1.1 to 31.4 and each averaged speed-up of 9.23 when one GetFeature request of the query

is cached. We can achieve speed-up ranging from 6.4 to 304.7 and each averaged speed-up of 73.25 when all GetFeature requests of the query are cached.

Table 3. The features involved in each query, where the columns *Streets*, *Roads*, *Places* are the number of instances retrieved from WFS servers for each type of feature.

Query	Streets	Roads	Places	Features after Spatial Join
1		29	9	(road: 10, place: 4)
2		1537	4	(road: 95, place: 4)
3	3449		9	(street: 360, place: 9)
4	2481		1	(street: 35, place: 1)
5	313		9	(street: 46, place: 4)
6	3449	5		(street: 349, road: 5)
7		9	4	(road: 2, place: 1)
8	3449	5	32	(street: 70, road: 5, place: 3)
9	8	3		(street: 8, road: 3)
10	42	2		(street: 42, road: 2)
11			1	(place: 1)

Table 4. Runtime performance of queries, where the columns *Time*, $Time^1$, and $Time^2$ are query runtime without caching, runtime with one cache hit (of the feature with the most number of instances), and runtime with cache hits of all features, respectively. The runtime includes the time for WFS queries, spatial join, and browser rendering. The last two columns show the size of the data retrieved from each WFS request and the corresponding time of server response and network transmission.

Query	Time (ms)	$Time^1$ (ms)	$Time^2$ (ms)	Data Size (kb)	Server/Network Time (ms)
1	314	124	18	40.4/1.3	214/63
2	1953	138	50	465/1.0	1550/88
3	5780	214	81	394/1.3	3900/97
4	4266	136	14	279/0.84	3007/94
5	881	127	21	34.9/1.3/0.63	670/72/67
6	7116	1425	1085	394/19.5	4160/283
7	437	104	10	24.9/1.0	393/191
8	6718	1357	1056	394/19.5/2.3	3810/200/166
9	234	134	3	2.2/1.3	201/113
10	348	139	6	5.9/5.2/0.63	123/300/93
11	277	253	2	0.63/0.63/0.83	165/258/141

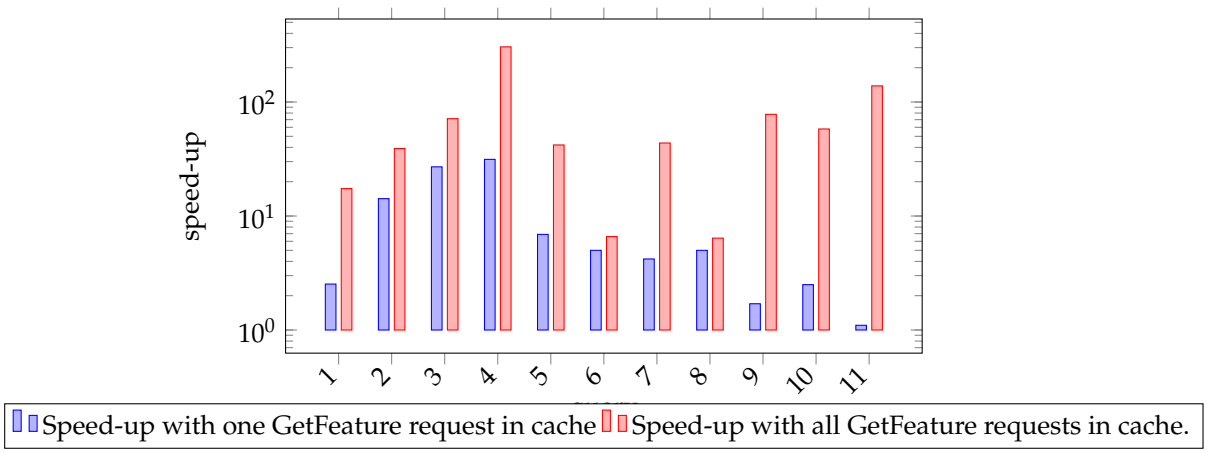


Figure 7. Speed-up (log-scale) of query response time via caching.

8. Discussion

Our algorithm uses mapping rules to translate RDF classes, properties, and values to the corresponding feature types, feature attributes, and attribute values. The mapping rules may be defined by someone who is familiar with the WFS data sources to associate low-level terminologies of data sources with higher-level concepts understood by common users. With the mapping rules, the algorithm can automatically translate queries formulated in higher-level concepts to WFS queries. The mapping rules only need to be defined once for each data source and the process is rather simple for a small number of feature types. However, if there are a large number of WFS feature types, it might be better to use an automatic matching algorithm to generate mapping candidates between WFS feature types and RDF classes and then manually finalize the mapping rules based on the generated candidates.

Our RDF query interface is able to catch errors in user queries during the process of query translation so that specific errors such as unknown RDF class or properties can be detected before any WFS GetFeature requests are sent. However, further improvement can be made to enhance the usability of the query interface. For example, a code completion feature can be added to the query interface to provide hints as users write their queries in a text box. The RDF mappings can also be externalized so that users can directly edit the mapping of RDF classes and properties to WFS feature types and attributes via Web interface.

9. Conclusions

In this paper, we present an algorithm to convert RDF queries to WFS requests so that users can query distributed WFS features as if they were RDF instances. The algorithm avoids the cost of converting features to RDF objects while retaining the benefits of RDF queries. User queries are written without specifying the underlying data sources or being concerned with joining the results from separate data sources.

We also demonstrated that query response time can be significantly reduced through dynamic caching of the WFS GetFeature results generated during the query processing. Many queries involve the same intermediate data requested from WFS servers. The ability to cache the results of WFS requests reduces not only query response time but also workloads of WFS servers and network traffic. Our experiments show that the time for WFS server response and network transmission is the major portion of the query runtime. User queries with cache hits can achieve a significant speed-up.

Acknowledgments: The publication cost was supported by the USA NSF [grant number 1414108].

Author Contributions: Tian Zhao developed the algorithm, implemented the prototype, performed the experiments, and wrote the paper. Chuanrong Zhang and Weidong Li conceived the original design, prepared the data, and have collaborated in prototype implementation and paper revision.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Battle, R.; Kolas, D. Enabling the geospatial Semantic Web with Parliament and GeoSPARQL. *Semantic Web* **2012**, *3*, 355–370.
2. Stadler, C.; Lehmann, J.; Hoffner, K.; Auer, S. LinkedGeoData: A core for a web of spatial open data. *Semantic Web* **2012**, *3*, 333–354.
3. GeoServer—Open Source Server for Sharing Geospatial Data. Available online: http://geoserver.org (accessed on 3 April 2017).
4. MapServer—Open Source Web Mapping. Available online: http://mapserver.org (accessed on 3 April 2017).
5. OpenLayers—A High-Performance, Feature-Packed Library for All Your Mapping Needs. Available online: http://openlayers.org (accessed on 3 April 2017).
6. SPARQL Query Language for RDF. Available online: https://www.w3.org/TR/rdf-sparql-query (accessed on 3 April 2017).

7. GeoSPARQL-A Geographic Query Language for RDF Data. Available online: http://www.opengeospatial.org/standards/geosparql (accessed on 3 April 2017).
8. Martin, D.; Paolucci, M.; Wagner, M. Toward semantic annotations of Web services: OWL-S from the SAWSDL Perspective. In Proceedings of the 6th International Semantic Web Conference, Busan, Korea, 11–15 November 2007; pp. 340–352.
9. Lausen, H.; Polleres, A.; Roman, D. Web Service Modeling Ontology (WSMO). Available online: http://www.w3.org/Submission/WSMO/ (accessed on 3 April 2017).
10. Fensel, D.; Bussler, C.; Ding, Y.; Omelayenko, B. The Web Service Modeling Framework (WSMF). *Electron. Commer. Res. Appl.* **2002**, *1*, 113–137.
11. Akkiraju, R.; Farrell, J.; Miller, J.; Nagarajan, M.; Schmidt, T.; Sheth, A. Web Service Semantics - WSDL-S. Available online: https://www.w3.org/Submission/WSDL-S/ (accessed on 3 April 2017).
12. Swift, T. An engine for computing well-founded models. *Lect. Notes Comput. Sci.* **2009**, *5649*, 514–518.
13. Alves, M.; Damasio, C.; Correia, N. RDF query inference in Prolog. *Commun. Comput. Inf. Sci.* **2016**, *649*, 191–201.
14. Papadimitriou, F. Artificial Intelligence in modelling the complexity of Mediterranean Ecosystems. *Comput. Electron. Agric.* **2012**, *81*, 87–96.
15. Shi, X.; Nellis, M.D. Semantic Web and service computation in GIScience applications: a perspective and prospective. *Geocarto Int.* **2014**, *29*, 400–417.
16. Yue, P. *Semantic Web-Based Intelligent Geospatial Web Services*; Springer Briefs in Computer Science; Springer: New York, NY, USA, 2013; p. 119.
17. Janowicz, K.; Scheider, S.; Pehle, T.; Hart, G. Geospatial semantics and linked spatiotemporal data: Past, present, and future. *Semantic Web* **2012**, *3*, 321–332.
18. Ashish, N.; Sheth, A., Eds. *Semantic Web and Beyond: Geospatial Semantics and the Semantic Web: Foundations, Algorithms, and Applications*; Springer: New York, NY, USA, 2011; p. 260.
19. Zhao, P.; Di, L.; (Eds). *Geospatial Web Services: Advances in Information Interoperability*; IGI Global: Hershey, PA, USA, 2011; p. 552.
20. Janowicz, K.; Raubal, M.; Levashkin, S. GeoSpatial Semantics. In Proceedings of the Third International Conference on GeoSpatial Semantics, GeoS 2009, Mexico City, Mexico, 3–4 December 2009.
21. Di, L.; Zhao, P. Geospatial Semantic Web interoperability. In *Encyclopedia of GIS*; Springer: New York, NY, USA, 2008; pp. 70–77.
22. Li, W.; Yang, C.; Raskin, R. A semantic enhanced model for searching in spatial web portals. In Proceedings of the AAAI Spring Symposium Semantic Scientific Knowledge Integration, Palo Alto, CA, USA, 26–28 March 2008; pp. 47–50.
23. Lutz, M.; Klien, E. Ontology-based retrieval of geographic information. *Int. J. Geogr. Inf. Sci.* **2006**, *20*, 233–260.
24. Kuhn, W. Geospatial semantics: Why, of what, and how? *J. Data Semant.* **2005**, *3*, 1–24.
25. Yue, P.; Di, L.; Yang, W.; Yu, G.; Zhao, P. Semantics-based automatic composition of geospatial web service chains. *Comput. Geosci.* **2007**, *33*, 649–665.
26. Wiegand, N.; Garcia, C. A task-based ontology approach to automated Geospatial Data Retrieval. *Trans. GISA* **2007**, *11*, 355–376.
27. Zhang, C.; Zhao, T.; W., L.; Osleeb, J. Towards logic-based geospatial feature discovery and integration using web feature service and geospatial semantic web. *Int. J. Geogr. Inf. Sci.* **2010**, *24*, 903–923.
28. Zhang, C.; Zhao, T.; Li, W. A framework for geospatial semantic web based spatial decision support system. *Int. J. Digit. Earth* **2010**, *3*, 111–134.
29. Zhang, C.; Zhao, T.; Li, W. Automatic search of geospatial features for disaster and emergency management. *Int. J. Appl. Earth Obs. Geoinf.* **2010**, *6*, 409–418.
30. Zhang, C.; Zhao, T.; Li, W. Towards improving query performance of Web Feature Services (WFS) for disaster response. *ISPRS Int. J. Geo-Inf.* **2013**, *2*, 67–81.
31. Zhao, T.; Zhang, C.; Wei, M.; Peng, Z.R. Ontology-based geospatial data query and integration. In *Geographic Information Science*; Lecture Notes in Computer Science LNCS5266; Springer: Berlin/Heidelberg, Germany, 2008; Volume 5266, pp. 370–392.
32. Zhao, T.; Zhang, C.; Anselin, L.; Li, W.; Chen, K. A parallel approach for improving geo-SPARQL query performance. *Int. J. Digit. Earth* **2014**, *8*, 383–402.

33. Li, W.; Yang, C.; Nebert, D.; Raskinc, R.; Houser, P.; Wu, H.; Z., L. A semantic-based web service discovery and chaining for building an Arctic spatial data infrastructure. *Comput. Geosci.* **2011**, *37*, 1752–1762.
34. Jones, J.; Kuhn, W.; Kebler, C.; Scheider, S. Making the web of data available via web feature services. In Proceedings of the 17th AGILE Conference on Geographic Information Science, Connecting a Digital Europe through Location and Place, Castellón, Spain, 3–6 June 2014.
35. Spanos, D.E.; Stavrou, P.; Mitrou, N. Bringing relational databases into the semantic web: A survey. *Semantic Web* **2012**, *3*, 169–209.
36. Tschirner, S.; Scherp, A.; Staab, S. Semantic access to INSPIRE how to publish and query advanced GML data. In Proceedings of the Workshop in Conjunction of 10th International Semantic Web Conference, Bonn, Germany, 23–27 October 2011.
37. LOD4WFS-Linked Open Data for Web Feature Services Adapter. Available online: http://github.com/jimjonesbr/lod4wfs (accessed on 3 April 2017).
38. Route Type Codes and Definitions. Available online: http://www.census.gov/geo/reference/rttyp.html (accessed on 3 April 2017).
39. Flexible RDF Query Interface for Distributed WFS Data. Available online: http://boyang.cs.uwm.edu:8080/newHaven/project.html (accessed on 3 April 2017).