Querying Distributed GIS with GeoPQLJ based on GeoJSON

Anna Formica, Mauro Mazzei, Elaheh Pourabbas and Maurizio Rafanelli
National Research Council, Istituto di Analisi dei Sistemi ed Informatica “Antonio Ruberti”,
Via dei Taurini, 19, I-00185, Rome, Italy

Keywords: GIS, Pictorial Queries, Geographic Operators, GeoJSON Specifications, Distributed Databases.

Abstract: Widespread use of the Web has increased the need to share and access distributed Geographic Information Systems (GIS). In this context, spatial query languages act as a guideline for Web-based GIS. In this paper, we focus on the Geographical Pictorial Query Language (GeoPQL) and enhance the related system in order to query distributed GIS. This is achieved by using the GeoPQLJ functions which are based on the GeoJSON format specifications. They are implemented in order to invoke the GeoPQL polygon-polyline operators, which is the focus of this paper. We define the logical diagram of the GeoPQLJ distributed system, and illustrate its underlying functionalities.

1 INTRODUCTION

In the area of Geographic Information Systems (GIS), visual spatial query languages are still a challenging topic. This arises from the need of providing the user with interactive and user-friendly tools for data manipulation and data retrieval, which are independent of the physical organization of data. Nowadays, an important requirement of these tools is also the enhancement of the interoperability across different systems in a world-wide distributed network, and the presence of native data models for querying topological relationships (Amirian et al., 2014). The usage of standard query languages for spatial data handling has been hindered by the lack of appropriate visual query languages, and hence the need for advanced geographic query languages in GIS has been emphasized since the nineties (Egenhofer, 1997).

Geographic queries, in general, can be better expressed by using graphical metaphors in query languages which are powerful to express the user’s mental model of the query (Kuhn, 1993). In GIS, geographic query languages should satisfy two basic requirements: they must be powerful and easy to use at the same time. Powerful, because they have to retrieve information about complex database schemas, keeping track of several relations existing among data. Easy to use, because the access to the stored information should not be limited to experts, but should be conceived for non-specialized end-users (Aoidh et al., 2008) (Kapler and Wright, 2005). These two basic requirements find a common solution in the development of advanced visual geographic query languages (Calcini and Mainguenaud, 1994). Visual query languages are high level declarative query languages, such as iconic (Lee and Chin, 1995), pictorial (Ferri and Rafanelli, 2005), or Query-by-Example (Papadis and Sellis, 1995), etc. They are user-friendly because their semantics is expressed by the user drawing itself, which specifies the properties that the query answer has to satisfy, and does not require the user to be aware about the underlying query language syntax.

In this paper, we concentrate on pictorial query languages, i.e., visual languages where queries are formulated by free-hand drawing (Formica et al., 2018). In the literature, some proposals have already discussed the way to formulate queries using pictorial configurations, for instance (Ferri and Rafanelli, 2005). In particular, in the mentioned paper a pictorial query language, called Geographical Pictorial Query Language (GeoPQL), has been proposed in order to address the user’s mental model of the query. It allows users to formulate their queries using drawing facilities and correctly interprets the query syntax and semantics on the basis of its underlying algebra. GeoPQL has been conceived as a stand-alone GIS client which uses the ESRI Library (Arcview, 2018).

We focus on GeoPQL and we enhance the related system in order to query distributed GIS by using open source JTS libraries (JTS, 2018). In this perspective, we selected GeoJSON because it is a simple
and text based format (GeoJSON, 2018). Since Geo-
JSON is a JSON encoding, the parsing and data inter-
changing for web services are flexible, and it is one
of the most popular encodings for transferring data to
client-side map visualization.

In the literature, beyond GeoJSON, the well-
known spatial formats are Shapefile, and Geogra-
phy Markup Language (GML) (Marquès-Mateu et al.,
2015). The Shapefile format is probably the most
common one in the fields of GIS and geomatics,
and was introduced by a commercial company (Esri,
2018). The GML format is a standard proposed by the
Open Geospatial Consortium (OGC) (OGC, 2018)
and essentially is an XML grammar for expressing ge-
ographical features whose specification is available in
several official documents. We selected GeoJSON be-
cause it is the most recent format, and is based on the
JavaScript object notation (JSON) (GeoJSON, 2018).
It is defined as a format for encoding a variety of ge-
ographic data structures, and is becoming a valid one
to send geographic data over computer networks or in
mobile devices (Sriparasa, 2013).

The main goal of this paper is to exploit the data
interchange property of GeoJSON to enhance the
GeoPQL system in order to pictorially query a network of distributed GIS. To this end, we intro-
duce the GeoPQLJ functions that are based on the
GeoJSON format specifications, as illustrated in Sec-
tion 4. These functions are defined for the polygon-
polyline topological relationships. In particular, the
operators we address in this paper are disjoint (DSJ),
inclusion (INC), touch (TCH), intersect (INT), and
pass-through (PTH). Successively, we enhance the
GeoPQL system to query and retrieve data in dis-
tributed GIS, and we describe the underlying logical
diagram. Then, we illustrate the functionalities of the
system by means of an experiment, where the results
of pictorial queries are given.

The paper is structured as follows. In the next sec-
tion, the related work is given, and in Section 3, an
overview about the GeoPQL operators is presented.
In Section 4, the GeoPQLJ functions are defined, and
in Section 5 the related distributed system is de-
scribed. In Section 6, the main functionalities of the
distributed system are illustrated. Finally, in Section
7 the conclusion is given.

2 RELATED WORK

Widespread use of the Web has increased the need to
share and access distributed GIS (Bo and Hui, 1999)
\footnote{A polyline is non self-intersecting (self-crossing), without loops, spirals, and bifurcations.}
(Liang et al., 2015) (Vatsavai, 2002). The current
Web-based GIS mostly adopt traditional client/server
or browser/server architectures (Liang et al., 2015).
In both these environments, the client/browser usu-
ally sends a request to the server, which processes it
and returns the result to the client. For this process,
spatial query languages essentially act as a guide-
line for Web-based GIS. As highlighted in (Amirian
et al., 2014), essentially two approaches are used for
handling spatial data. The first approach is to use
the specifications published and managed by OGC,
whereas the second one is to use web services as core
technologies, such as XML, XSD, WSDL, SOAP. The
main geospatial web service is Web Feature Service
(WFS), which provides access to geospatial data us-
ing GML format. This format contains both geomet-
rical and attribute properties of data.

For instance in (Vatsavai, 2002), the authors pro-
pose the spatial query language GML-QL derived
from XQuery, which is essentially based on two types
of queries, i.e., unary and binary types. They also de-
fine some spatial functions and operations to support
spatial queries. Similarly, in (Guan et al., 2006), the
authors propose GQL, a query language specification
to support spatial queries over GML documents by
extending the data model, algebra, and semantics of
XQuery.

In (Almendros-Jimenez et al., 2015), an XQuery
library for querying Open Street Map (OSM) has been
proposed, following the approach given in (Boucelma
and Colonna, 2004). This library is based on the
spatial operators originally introduced in (Clementini
et al., 1993) and (Egenhofer, 1991). In (Huang et al.,
2009), a similar proposal has been defined which is
based on GML, and also addresses the performance
problem of using XML/GML-native technologies in
order to manipulate large GML documents.

The aforementioned papers present an approach
similar to our proposal, but the former is limited to the
OSM environment, whereas the latter addresses a
general distributed environment without supporting
functional operators. With respect to these papers, our
approach can be used in a general distributed system
thanks to GeoJSON which, by making use of spatial
operators, allows the access to geo-spatial reposito-
ries in a more efficient way. In addition, with respect
to (Almendros-Jimenez et al., 2015), where data are
extracted according to a composition and filtering ap-
proach, in our proposal the user is not required to be
aware about complex query syntaxes because he/she
can benefit of the pictorial querying facilities. In fact,
we use GeoPQL (Ferri and Rafanelli, 2005), and en-
hance the related system in order to query distributed
GIS by using open source JTS libraries (JTS, 2018).
3 GeoPQL OPERATORS:
OVERVIEW

In this paper, we focus on GeoPQL which is based on the notion of Symbolic Graphical Objects (SGO) (Ferri et al., 2002), (Formica et al., 2013). It has been defined to graphically represent the spatial configurations of geographic entities (i.e., point, polyline, and polygon), and the spatial relationships between SGO.

Definition 3.1 \([SGO]\). Given a GIS, a Symbolic Geographical Object \((SGO)\) is a 5-tuple \(\psi = (id, \text{geometric type}, \text{object class}, \Sigma, \Lambda)\) where:

- \(id\) is the \(SGO\) identifier assigned by the system to uniquely identify the \(SGO\) in a query;
- \(\text{geometric type}\) can be a point, a polyline or a polygon;
- \(\text{object class}\) is the geographical concept name belonging to the database schema and iconized by the \(SGO\), identifying a geographical class (set of instances) of the database;
- \(\Sigma\) represents the set of typed attributes of the \(SGO\) which can be associated with a set of values by the user;
- \(\Lambda\) is an ordered set of pairs of coordinates, which defines the spatial extent and position of the \(SGO\) with respect to the coordinate reference system of the working area.

The GeoPQL algebra consists of a set of binary geo-operators, which are logical (Geo-union, Geo-any, Geo-alias), metrical (Geo-difference, and Geo-distance), and topological (Geo-disjunction, Geo-touching, Geo-inclusion)\(^2\), Geo-intersect, Geocrossing, Geo-pass-through, Geo-overlapping, Geo-equality). Our focus, as mentioned in the Introduction, is on the polygon-polyline topological relationships, therefore, in this work we consider a subset of the topological operators, namely, Geo-disjunction (DSJ), Geo-touching (TCH), Geo-inclusion (INC), Geo-intersect (INT), and Geo-pass-through (PTH). Indeed, the remaining operators are not considered because in the case of the polygon-polyline relationship they are not applicable (for instance Geocrossing operator which is defined between polylines, Geo-overlapping which is defined between polygons, or Geo-equality which is defined between polylines or between polygons).

\(^2\)Note that in our approach the operators \textit{cover} and \textit{covered-by}, extensively used in the literature, can be represented by using the Geo-inclusion operator.

Definition 3.2 [Geo-operators]. Given a polygon \(\mathcal{P}\), and a polyline \(\mathcal{L}\), which are two \(SGO\). Let \(i, b\), and \(e\) denote, respectively, the \textit{interior}, \textit{boundary}, and \textit{exterior} points of the \(SGO\). Then, the binary geo-operations DSJ, INC, TCH, INT, and PTH, are formally defined as follows, where \(k, j \in \{i, b, e\}\):

- DSJ (geo-disjunction):
  \[\mathcal{P} \text{ DSJ } \mathcal{L} \iff \mathcal{P}_i \cap \mathcal{L}_j = \emptyset, k \neq e\]

- INC (geo-inclusion):
  \[\mathcal{P} \text{ INC } \mathcal{L} \iff \mathcal{P}_k \cap \mathcal{L}_j = \mathcal{L}_j, j, k \neq e\]

- TCH (geo-touching):
  \[\mathcal{P} \text{ TCH } \mathcal{L} \iff \exists x \in \mathcal{L}_b \cap \mathcal{P}_b, I(x), \text{ that is a neighborhood of } x, \text{ and only one of the following holds:} I(x) \cap \mathcal{L}_j \cap \mathcal{P}_e = \emptyset \text{ or } I(x) \cap \mathcal{L}_j \cap \mathcal{P}_i = \emptyset, \text{ where } j \neq e\]

- INT (geo-intersect):
  \[\mathcal{P} \text{ INT } \mathcal{L} \iff \mathcal{P}_k \cap \mathcal{L}_i \neq \emptyset, k = i, e\]

- PTH (geo-pass-through):
  \[\mathcal{P} \text{ PTH } \mathcal{L} \iff \mathcal{P}_k \cap \mathcal{L}_i \neq \emptyset, k = i, e \text{ and } \mathcal{P}_e \cap \mathcal{L}_b \neq \emptyset\]

Note that in the above definition, the order of the operands is not relevant. The geo-operators are invoked in the GeoPQLJ functions, which are introduced in the next section.

4 GeoPQLJ FUNCTIONS

In this section, we introduce the GeoPQLJ functions, that are based on the GeoJSON format specification. GeoJSON is a format for encoding a variety of geographic data structures using JavaScript Object Notation (JSON) (Bray, 2014). A GeoJSON object may represent a region of space (Geometry), a spatial entity (Feature), or a list of Features (FeatureCollection). It supports the following geometry types: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection.

The geometry types of GeoJSON are defined in the OpenGIS Simple Features Implementation Specification for SQL (SFSQL). They are: 0-dimensional Point, and MultiPoint; 1-dimensional curve LineString, and MultiLineString; 2-dimensional surface Polygon, and MultiPolygon; and the heterogeneous GeometryCollection. GeoJSON representations of instances of these geometry types are analogous to the well-known text (WKT) and well-known binary (WKB) representations, originally defined by OGC (OGC, 2018). In particular, the former is a text markup language for representing geometry objects according to a vector format and reference systems of spatial objects. The latter is used...
to transfer and store the same information in specific geographic databases.

The GeoPQLJ functions have been inspired by taking into account the syntax of *Turf.js* (Turf, 2018). In the following, the spatial types *Point*, *LineString*, and *Polygon*, which allow us to define the spatial operators’ functions, are given. These operators are: *disjoint*, *inclusion*, *touch*, *intersect*, and *passsthrough*, which are admissible for representing the topological relationships between *Polygon* and *LineString*.

```javascript
const point = {  
    "type": "Feature",  
    "properties": ,  
    "coordinates": [x, y]  
};

const line = {  
    "type": "LineString",  
    "coordinates": [[x1, y1], [x2, y2], ..., [xn, yn]]  
};

const polygon = {  
    "type": "Polygon",  
    "coordinates": [[x1, y1], [x2, y2], ..., [xn, yn], [x1, y1]]  
};
```

In the following functions, *SGO* and *SGO* are generalizations of the geometric types defined above:

1. declare function *GeoPQLJ.disjoint*(SGO1, SGO2) (*GeoPQLJ*booleanQuery(SGO1, SGO2, "GeoPQLJ")

where the function *disjoint* is defined as follows:

```javascript
function disjoint(geom1, geom2) {  
    switch (geom1.type) {  
        case "LineString":  
            switch (geom2.type) {  
                case "Polygon":  
                    return false;  
                }  
            }  
        case "Polygon":  
            switch (geom2.type) {  
                case "LineString":  
                    return false;  
                }  
            }  
        break;  
    }
```

2. declare function *GeoPQLJ.inclusion*(SGO1, SGO2) (*GeoPQLJ*booleanQuery(SGO1, SGO2, "GeoPQLJ")

where the function *inclusion* is defined as follows:

```javascript
function inclusion(geom1, geom2) {  
    switch (geom1.type) {  
        case "LineString":  
            switch (geom2.type) {  
                case "Polygon":  
                    return true;  
                }  
            }  
        case "Polygon":  
            switch (geom2.type) {  
                case "LineString":  
                    return true;  
                }  
            }  
        break;  
    }
```

3. declare function *GeoPQLJ.touch*(SGO1, SGO2) (*GeoPQLJ*booleanQuery(SGO1, SGO2, "GeoPQLJ")

where the function *touch* is defined as follows:

```javascript
function touch(geom1, geom2) {  
    switch (geom1.type) {  
        case "LineString":  
            switch (geom2.type) {  
                case "Polygon":  
                    return isPointOnLineEnd(geom1, geom2);  
            }  
        break;  
    }
```

4. declare function *GeoPQLJ.intersect*(SGO1, SGO2) (*GeoPQLJ*booleanQuery(SGO1, SGO2, "GeoPQLJ")

where the function *intersect* is defined as follows:

```javascript
function intersect(geom1, geom2) {  
    switch (geom1.type) {  
        case "LineString":  
            switch (geom2.type) {  
                case "Polygon":  
                    return true;  
            }  
        break;  
    }
```

5. declare function *GeoPQLJ.passsthrough*(SGO1, SGO2) (*GeoPQLJ*booleanQuery(SGO1, SGO2, "GeoPQLJ")

where the function *passsthrough* is defined as follows:

```javascript
function passsthrough(geom1, geom2) {  
    switch (geom1.type) {  
        case "LineString":  
            switch (geom2.type) {  
                case "Polygon":  
                    return true;  
            }  
        break;  
    }
```

Table 1 summarizes the above mentioned functions with the corresponding brief definitions and invoked GeoPQL operators, where “Ret. T” stands for “Returns True”.

5. THE GeoPQLJ DISTRIBUTED SYSTEM

In Figure 1, the GeoPQLJ distributed system is illustrated, where the dashed box represents the server side of the system. As shown in the figure, within the local GeoPQL system the user expresses his/her pictorial query, which is transformed into an SQL
like query, by identifying the corresponding operator, GeoPQL\_op, as follows:

\[ \text{geom1 GeoPQL\_op geom2} \]

where geom1 and geom2 are two SGO.

Successively, the SQL like query is translated into the format defined according to the GeoPQLJ library described in the previous section and a GeoPQLJ query is generated. This query is sent to a Web Service, in our case the Web Feature Service (WFS) of the Open Geospatial Consortium (OGC, 2018), which connects to the data sources in order to select the required URLs. In other words, the WFS provides a data format, e.g., Geography Markup Language (GML), GeoJSON, etc., that allows us to access the territorial data directly from distributed data sources, as well as to analyze and process them. In this phase, the required URLs are retrieved by accessing the GeoPortal the user is interested in, where the list of all possible available open services are shown.

Analogously to the other web services, the WFS uses the well-known methods GetCapability, GetMap, and GetFeatureInfo. These methods capture the general information about the metadata that contains the list of features corresponding to the query. This list is associated with the Geodetic Parameter Registry (EPSG), that allows the accurate overlapping of the required features.

In order to access the metadata related to the retrieved features, the GetFeature is applied to each feature. In particular, the system associates, on the basis of the GeoJSON specification, the corresponding spatial geometries (geom1, geom2) with the retrieved features. The GetFeature allows us to access the content (GeoData) of the features’ attributes, and the related spatial coordinates, which are distributed on different sites. In particular the following method is invoked:

\[
\text{GeoJSON\_op(geom1\_from\_wkt('Polygon([\{x_1,y_1\}, [x_2,\ldots], [x_n, y_n]\}]), geom2\_from\_wkt('LineString([\{x_1, y_1\}, [x_2,\ldots], [x_n, y_n]\}]) \rightarrow true)}
\]

where GeoJSON\_op corresponds to a generic GeoJSON operator, and the wkt method converts the geometries to the Well-Known Text (WKT) geometry formats (Turf, 2018). In the wkt method, the WKTReader extracts the geometry objects from either Readers (i.e., the abstract class for reading character streams) or Strings. It is a parser that allows us to read the geometry objects from text blocks embedded in other data formats (e.g., XML). Finally, the GetMap request is sent to the Web Client, and the final query answer is shown as a map.
6 ILLUSTRATION OF THE SYSTEM

In order to better show the proposed GeoPQL system, in this section an experiment is given. It has been realized by accessing the data sources from the site (Catalog, 2018), which publishes and manages datasets, as for instance the U.S. Geological Survey or NASA. In particular, let us suppose the user is interested in the following query:

"Find the roads which pass through national parks in New Mexico"

He/she formulates it by drawing a pictorial query in the Local GeoPQL System as shown in Figure 2. In this case the pictorial query is essentially given by a polygon passed through by a polyline which will be associated, respectively, with the national parks and the roads of New Mexico as shown below. This query is transformed into the following SQL like query:

\[ geom_1 \text{ PTH } geom_2 \]

where \( geom_1 \) and \( geom_2 \) are a polyline and a polygon, respectively, (or viceversa) and \( PTH \) is the pass through geo-operator. In turn, this query is transformed into the following query:

\[ geom_1 \text{ geopql j_pth } geom_2 \]

In this phase, the system associates, on the basis of GeoJSON specification, the corresponding spatial geometries (\( geom_1 \), \( geom_2 \)) with the retrieved features (i.e., \( GPS \) Roads and \( National \) Park Boundaries). In particular, the WFS service connects to the data sources by means of the links given below. The first link:

https://catalog.data.gov/dataset/gps-roads

refers to all interstate highways, US highways, most of the state highways, and some county roads in New Mexico. These data are represented according to a 1:100,000 scale. Furthermore, the other link:

https://catalog.data.gov/dataset/national-park-boundariesf0a4c

refers to the National Park Boundaries in New Mexico. In Figure 3 the geodata we are interested in, both roads and national parks in New Mexico, are shown.

This query is then translated into GeoPQLJ syntax and it is elaborated in the distributed system by accessing the above mentioned links (see as Figure 1). To this end, the features \( GPS \) Roads and \( National \) Park Boundaries are retrieved by applying the GetCapability, GetMap, and GetFeatureInfo methods.

The answer to the query is delivered to the web client and visualized as a map to the user. In particular, the roads which pass through the national parks in New Mexico are 32. A fragment of the answer pro-
Analogously, the user might be interested to know the roads that are disjoint from, are included in, touch, or intersect the national parks. In the experiment, the total number of features involved in the queries are 11299 roads and 510 parks. In Table 2, in the second column the answers to these queries are shown, whereas in the third column the execution times (in seconds) are given. Note that, in the case of the disjoint (geopql\_ds) and the intersect (geopql\_int) operators, the numbers of selected objects are 11256 and 41, respectively, whose sum corresponds to the total of 11299 roads. We observe that in the case of the touch operator (geopql\_tch), the result is equal to zero due to the scale of the object representation. Furthermore, the execution time related to it is higher with respect to the others because of its more complex semantics, as shown in Section 3 (see Definition 3.2).

In Figures 5, 6, 7, and 8, fragments of the answers related to the DSJ, INC, TCH, and INT are shown, respectively.

7 CONCLUSION

In this paper we presented the GeoPQLJ distributed system for pictorially querying distributed GIS. We introduced the GeoPQLJ functions which are based on the GeoJSON format specifications. They have been defined for the GeoQL polygon-polyline operators, which is the focus of this paper.
As a future work, we are planning to investigate the problem of query constraint relaxation in distributed GIS, in order to provide approximate answers to queries.

REFERENCES


182