Querying Open Street Map with XQuery

Jesús M. Almendros-Jiménez and Antonio Becerra-Terón
Information Systems Group, University of Almería, 04120 Almería, Spain
{*jalmen, abecerra*}*@ual.es*

Keywords: Open Street Map, Urban Maps, Spatial Databases, XQuery, XML.

Abstract: In this paper we present a library for querying Open Street Map (OSM) with XQuery. This library is based on the well-known spatial operators defined by Clementini and Egenhofer, providing a repertoire of XQuery functions which encapsulate the search on the XML document representing a layer of OSM, and make the definition of queries on top of OSM layers easy. In essence, the library provides a repertoire of OSM Operators for points and lines which, in combination with Higher Order facilities of XQuery, facilitates the composition of queries and the definition of keyword based search geo-localized queries. OSM data are indexed by an R-tree structure, in which points and lines are enclosed by Minimum Bounding Rectangles (MBRs), in order to get shorter answer time.

1 INTRODUCTION

Open Street Map (OSM) (Haklay and Weber, 2008) is a collaborative project to create a free editable map of the world. It is supported by the non-profit organization called *OSM Foundation*. OSM data can be visualized from the OSM web site[1], but also many applications have been built for the handling of maps (see http://wiki.openstreetmap.org/wiki/Software for a list of tools). OSM can be represented with many formats; in fact, there are many tools available in order to export OSM to XML, KML, SVG, etc.

With the increasing interest in OSM, many tools have been devoloped. However, their main taks are edition, export, rendering, conversion, analysis, routing and navigation, and little attention focuses on querying. Querying urban maps can be seen from many points of view. One of the most popular querying mechanism is the so-called *routing* or *navigation*; for instance, the most suitable route to go from one point to another of the city. In this case, the input of the query are two points (or streets) and the output is the sequence of instructions needed to reach the destination.

Nevertheless, querying an urban map can also be interesting for city sightseeing. In this case, places of interests around a given geo-localized point are the major goal. The input of the query are a point and a city area, close to the given point, and the output is a set of points. The tourist would also like to query streets close to a given street when looking for a hotel, querying parking areas, restaurants, high ways to go out, etc. In such queries, the input is a given point (or street) and the output could be a number of streets, parking areas, restaurants, high ways, etc.

Most tools are able to query OSM with very simple commands: searching by tag and relation names. This is the case of *JOSM*[2] and *Xapiviewer*[3]. The *OSM Extended API or XAPI*[4] is an extended API that offers search queries in OSM with a XPath flavoring. The *Overpass API (or OSM3S)*[5] is an extension to select certain parts of the OSM layer. Both XAPI and OSM3S act as a database over the web: the client sends a query to the API and gets back the dataset that corresponds to the query. *OSM3S* has a proper query language which can be encoded by an XML template. *OSM3S* offers more sophisticated queries than *XAPI*, but it is equipped with a rather limited query language.

XQuery (Robie et al., 2014; Bamford et al., 2009) is a programming language proposed by the W3C as standard for handling XML documents. It is a functional language in which *for-let-orderby-where-return (FLOWR)* expressions are able to traverse XML documents. It can express Boolean conditions, and provides a format to output documents. XQuery has a sublanguage, called *XPath* (Berglund et al.,

[1]http://www.openstreetmap.org

[2]https://josm.openstreetmap.de/

[3]http://osm.dumoulin63.net/xapiviewer/

[4]http://wiki.openstreetmap.org/wiki/Xapi

[5]http://overpass-api.de/

Almendros-Jiménez J. and Becerra-Terón A..
Querying Open Street Map with XQuery.
DOI: 10.5220/0005365500610071
In *Proceedings of the 1st International Conference on Geographical Information Systems Theory, Applications and Management* (GISTAM-2015), pages 61-71
ISBN: 978-989-758-099-4

2010), whose role is to address nodes on the XML tree. XPath is properly a query language equipped with Boolean conditions and many path-based operators. XQuery adds expressivity to XPath by providing mechanisms to join several XML documents.

In this paper, we present a library for querying OSM with XQuery. This library is based on the well-known spatial operators defined by Clementini (Clementini and Di Felice, 2000) and Egenhofer (Egenhofer, 1994), providing a repertoire of XQuery functions which encapsulate the search on the XML document representing a layer of OSM, and making the definition of queries on top of OSM layers easy. Basically, the library provides a repertoire of *OSM Operators*, for points and lines which, in combination with *Higher Order* facilities of XQuery, makes the *Composition of Queries* and the definition of *Keyword based* search *Geo-Localized* queries easy. OSM data are indexed by an R-tree structure (Hadjieleftheriou et al., 2008), where lines and points are enclosed by *Minimum Bounding Rectangles (MBRs)* in order to get shorter answer time.

Our work focuses on the retrieval of information and querying from urban maps. Although navigation is a interesting type of query, we are more interested in querying the elements of a urban map in a certain area or layer and taking as input a given point or street. Queries about buildings, parkings, lakes, etc. is considered as future work. The advantages of our approach are that our XQuery library makes the definition of queries on top of OSM layers easier. A repertoire of OSM spatial operators are implemented in terms of the spatial operators of Clementini and Egenhofer. Such repertoire of operators is specific for OSM maps, that is, it handles the particular nature of the XML representation of OSM. It includes, for instance, the operator *isEndingTo* for streets (i.e. *ways*), which returns *true* whenever a street ends (without crossing) to another one. Another operator, *isContinuationOf* also for streets, returns *true* whenever a street is the continuation of another one. Both are particular cases of Clementini's operator *touches*. In addition, our proposal includes a batch of emphCoordinate based XQuery functions, allowing the expression of interesting *Geo-positioning* queries. Higher order functions in XQuery[6] allow definitions of composition of queries and keyword based search queries in a easy way. Queries are expressed in terms of filtering, composition, set-based operators (union, intersection and difference) as well as mapping.

For instance, a typical query in our approach is something like: "Retrieve the schools close to a street, wherein *"Calzada de Castro"* street ends" which combines proximity to a street, keywords (i.e., *school*), as well as the operator (i.e., *isEndingTo*). It can be expressed as follows:

```
let $waysAllEndingTo :=
fn:filter(
rt:getLayerByName(.,"Calle Calzada de Castro"),
osm:isEndingTo(osm:getOneWay(., "Calle Calzada de Castro"),?))
return
fn:filter(
fn:for-each($waysAllEndingTo, rt:getLayerByOneWay(.,?)),
osm:searchTags(?,"school"))
```

which uses higher-order functions (i.e., *filter* and *for-each*) of XQuery.

A good performance of query processing is ensured due to the use of indexing for OSM data. An R-tree structure implemented as an XML document is used to index OSM nodes and ways enclosed by MBRs. Using the R-tree structure, we are able to retrieve the elements (i.e., points and streets) *close* to a given point or street, and thus, to process in reasonable time, queries focused on the vicinity of a point or street even for large city maps. Thus, for *Geo-localized Queries*, we can get better answer times.

We have implemented our library with the *BaseX* XQuery processor (Grun, 2015). The implementation is based on the transformation of geometric shapes of OSM into the corresponding GML data. Then GML data are handled by the *Java Topology Suite (JTS)* (Shekhar and Xiong, 2008), an open source API that provides a spatial object model and a set of spatial operators. JTS is available for most of XQuery processors due to the *XQuery Java Binding* mechanism. This is the case of *Exist* (Meier, 2003) and *Saxon* (Kay, 2008) processors as well as *BaseX*. Thus, the library is portable to other XQuery implementations. We have also tested our approach by using the *JOSM* tool (Haklay and Weber, 2008), that works with the XML representation of OSM data, customized with an style to highlight points and streets obtained from the queries. We have evaluated our library with datasets of several sizes, for which benchmarks show that shorter answer times are obtained even for large city maps. Finally, the developed library is available from http://indalog.ual.es/osm. The examples shown in this paper can also be downloaded here.

The rest of this article is organized as follows. Section 2 will present the basic elements of Open Street Map. Section 3 will define the XQuery library. Section 4 will show examples of queries and give benchmarks for several datasets. Section 5 will compare with related work and finally, Section 6 will conclude and present future work.

[6]http://www.w3.org/TR/xpath-functions-30/#higher-order-functions

2 OPEN STREET MAP

OpenStreetMap uses a topological data structure which includes the following core elements: (1) *Nodes* which are points with a geographic position, stored as coordinates (pairs of a latitude and a longitude) according to WGS84. They are used in ways, but also to describe map features without a size like points of interest or mountain peaks. (2) *Ways* are ordered lists of nodes, representing a poly-line, or possibly a polygon if they form a closed loop. They are used in streets and rivers as well as areas: forests, parks, parkings and lakes. (3) *Relations* are ordered lists nodes, ways and relations. Relations are used for representing the relationship of existing node points and ways. (4) *Tags* are key-value pairs (both arbitrary strings). They are used to store *metadata* about the map objects (such as their type, their name and their physical properties). Tags are attached to a node, a way, a relation, or to a member of a relation.

As an example of OSM, Figure 1 shows the visualization with JOSM of a piece of the Almería (Spain) city map. In order to represent such a map, OSM uses XML labels: *node*, *relation* and *way*, and each label can have several attributes; for instance, *node* has *lat* and *lon*, among others, for representing latitude and longitude of the node. A node, representing a point of interest of the city, can have *tags* for adding information about the point, using attribute pairs key (k) and value (v) with this end. For instance, the museum "Museo Arqueologico" of Almería city is represented as follows:

```
<node lat='36.8386557' lon='-2.4556049'>
    <tag k='name' v='Museo Arqueologico' />
    <tag k='tourism' v='museum' />
</node>
```

The main element of the OSM is the *way* that serves not only to represent streets but also buildings, parkings, etc. Ways are described by a sequence of node references, called *nd*, which link ways to nodes, and *tag*s as follows:

```
<way>
    <nd ref='-3625' />
    <nd ref='-3623' />
    <nd ref='-3621' />
    <tag k='highway' v='residential' />
    <tag k='name' v='Calle Calzada de Castro' />
</way>
```

When the way is related to a building, park, etc, specific tags are used inside the way description, for instance:

```
<way id='27161540'>
    <nd ref='298004115' />
    <nd ref='298004116' />
    <nd ref='298004119' />
    <nd ref='298004128' />
    <nd ref='298004115' />
    <tag k='amenity' v='parking' />
</way>
```

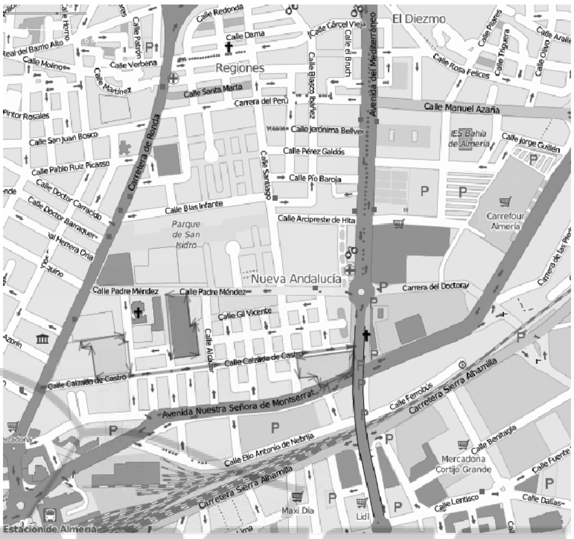


Figure 1: (Spain) Almería City Map.

In spite of the simplicity of the XML representation of OSM, many features in a OSM layer (see http://wiki.openstreetmap.org/wiki/Map_Features for a list) can be described. Finally, relations are used to relate elements of the map, for instance, bus routes:

```
<relation id='147091'>
    <member type='way' ref='27197940' role='3,11,12' />
    <member type='way' ref='27197939' role='3,7,11,12' />
    <member type='way' ref='35031199' role='3,11,12' />
    <member type='way' ref='27197944' role='7' />
    <member type='way' ref='27197945' role='7' />
    <member type='way' ref='25586878' role='3,11,12' />
    <member type='way' ref='30953417' role='3,11,12' />
    <member type='way' ref='25585669' role='3,5,6,11,12' />
    <member type='way' ref='27161590' role='5,6,12' />
    <member type='way' ref='27210271' role='3' />
    <member type='way' ref='31484654' role='12' />
    <member type='way' ref='27210293' role='3,5,6,12' />
    <member type='way' ref='50004718' role='12' />
<tag k='route' v='bus' />
<tag k='type' v='route' />
</relation>
```

3 XQuery LIBRARY FOR OSM

Our main goal is to provide a repertoire of OSM Operators, implemented as a XQuery library which, in combination with Higher Order facilities of XQuery, enables the expression of spatial queries over OSM maps easily. Moreover, we have to ensure shorter answer time for large maps. An R-tree structure to index OSM maps has been implemented, and suitable XQuery functions to retrieve the layer of objects close to a given node and way have been developed.

Next, we will show the elements of the XQuery library which includes:

(1) *OSM Indexing* to generate an R-tree and retrieve elements from it,

(2) *Transformation Operators* to transform OSM geometries into GML ones,

(3) *OSM Spatial Operators* to check spatial relations over OSM geometries, that is, ways and nodes representing streets and points, respectively,

(4) *Higher Order functions* to facilitate the composition of queries and keyword based search queries.

3.1 OSM Indexing

In order to handle large city maps, in which the layer can include many objects, an R-tree structure to index objects is used. The R-tree structure is based, as usual, on MBRs to hierarchically organize the content of an OSM map. Moreover, they are also used to enclose the nodes and ways of OSM in leaves of such structure. Figure 2 shows a visual representation of the R-tree of a OSM layer for Almería (Spain) city map. These ways have been highlighted in different colors (red and green) and MBRs are represented by light green rectangles.

The R-tree structure has been implemented as an XML document. That is, the tag based structure of XML is used for representing the R-tree with two main tags called *node* and *leaf*. A node tag represents the MBR enclosing the children nodes, while leaf tag contains the MBR of OSM ways and nodes. The tag *mbr* is used to represent MBRs. For instance, the R-tree of the OSM map of Figure 1 is represented in XML as follows:

```
<node x="-2.4574724" y="36.8305714"
z="-2.4473768" t="36.849285">
  <node x="-2.4565026" y="36.8319462"
   z="-2.4476476" t="36.849285">
    <node x="-2.4557511" y="36.8319462"
     z="-2.4491401" t="36.8414807">
      <leaf x="-2.4557511" y="36.8347249"
       z="-2.4522051" t="36.8396123">
        <mbr x="-2.4533564" y="36.8383646"
         z="-2.452359" t="36.8384662">
            <way ...>
            ....
            </way>
        </mbr>
            ....
```

The root element of the XML document is the root node of the R-tree, and the children can be also nodes

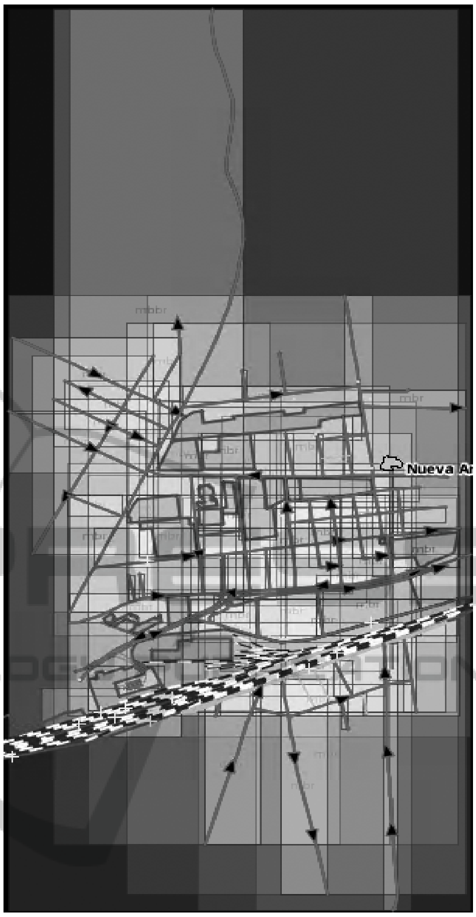


Figure 2: R-tree based indexing of OSM Maps.

and, in particular, leaves. x, y, z and t attributes of nodes are the left (x,y) and right corners (z,t) of the MBRs. MBRs are also represented by left and right corners.

We have implemented in XQuery a set of functions to handle R-trees for OSM. The function `load_file` generates a R-tree from an OSM layer. The function `getLayerbyName` obtains, given the name of a node or way, the nodes or ways of the OSM layer whose MBR overlaps the MBRs of the given node or way. In case of points, overlapping means inclusion. In other words, `getLayerbyName` obtains the elements that are *close* to the given node or way. Finally, each node and way in isolation can be retrieved by means of `getOneWay` and `getNode`, respectively.

Our proposed query language uses `getLayerbyName` as basis, in the sense that, queries have to be related to a certain area of interest, given by the name of a point (park, pharmacy, etc.,) or by the name of a street. In other words, our query language is useful for *Geo-localized queries*. Once the layer of the area of interest is retrieved,

Name	*Definition*
Equals(a,b)	Their interiors intersect and no part of the interior or boundary of one geometry intersects the exterior of the other
Disjoint(a,b)	They have no point in common
Touches(a,b)	They have at least one boundary point in common, but no interior points
Contains(a,b)	No points of b lie in the exterior of a, and at least one point of the interior of b lies in the interior of a
Covers(a,b)	Every point of b is a point of (the interior of) a
Crosses(a,b)	They have some but not all interior points in common (and the dimension of the intersection is less than at least one of them)
Overlaps(a,b)	They have some but not all points in common, they have the same dimension, and the intersection of the interiors of the two geometries has the same dimension as the geometries themselves

Figure 3: Clementini Spatial Operators.

the repertoire of OSM operators in combination with Higher Order functions can be applied to produce complex queries. The answer of a query will be an OSM layer including points and streets of the area of interest.

3.2 Transformation Operators

In order to handle OSM entities (i.e., nodes and ways), OSM geometries of these entities have to be transformed into GML data. Once transformed, the GML data will be handled by the JTS library based on Clementini's operators. In our case, functions `_osm2GmlLine` and `_osm2GmlPoint` have been defined, in order to transform OSM *ways* and *nodes* into GML multi-lines and points, respectively. *_osm2GmlPoint* is defined as follows:

```
declare function osm_gml:_osm2GmlPoint($node as node()){
  <gml:Point>
  <gml:coordinates>
  {
  let $lat := $node/@lat, $lon := $node/@lon
  return (concat(concat(data($lat),',' ),data($lon)))
  }
  </gml:coordinates>
  </gml:Point>};
```

3.3 OSM Spatial Operators

A repertoire of *OSM Operators* suitable for OSM city maps has been designed. That repertoire is specific for OSM maps which means that it handles the particular nature of the XML representation of OSM whose basis is the well-known spatial operator proposal defined by Clementini (Clementini and Di Felice, 2000). and represented in Figure 3[7]. We can see in Figures 4 and 5 our proposal of (Boolean) *OSM Operators* for querying maps. We can consider two kinds of operators:

(a) *Coordinate based OSM Operators*, shown in Figure 4;

(b) *Clementini based OSM Operators*, shown in Figure 5.

They are designed to cover most of urban queries involving points (i.e., nodes) and streets (i.e., ways). Usually, we would like to express queries related to geo-positioning, i.e., streets at north, points at east, and so on; the street in which a given point is located; if two points are located in the same street; whether two streets are crossing in any point or not; whether a street ends to another one, and finally, whether a street is a continuation of another one.

Next, we will show the implementation of our OSM Operators. For instance, the coordinate based operator *furtherNorthPoints*, which is *true* whenever the first point is further north than the second point, is defined as follows:

```
declare function osm:furtherNorthPoints($node1 as node(),
$node2 as node())
{
  let $lat1 := $node1/@lat, $lat2 := $node2/@lat
  return
  (: Case 1:  both nodes in positive Ecuador hemisphere :)
    if ($lat1 > 0 and $lat2 > 0) then
        if (($lat2 - $lat1) > 0) then true()
                                 else false()
    else
  (: Case 2: both nodes in negative Ecuador hemisphere :)
      if ($lat1 < 0  and $lat2 < 0) then
      if (((-$lat2) - (-$lat1)) < 0) then true()
                                     else false()
      else
  (: Case 3: First node in positive Ecuador hemisphere,
            Second node in negative Ecuador hemisphere:)
     if ($lat1 > 0 and $lat2 < 0) then false()
  (: Case 4: First node in negative Ecuador hemisphere,
           Second node in positive Ecuador hemisphere :)
                                  else true()
};
```

[7]Clementini has also defined the logic negation of some operators, that is, *Intersects* (for *Disjoint*), *Within* (for *Contains*) and CoveredBy (for *Covers*).

Name	*Definition*	*Spatial Op.*
furtherNorthPoints(p1,p2)	Returns true whenever p1 is further north than p2	Using latitudes by considering points in north and south hemispheres
furtherSouthPoints(p1,p2)	Returns true whenever p1 is further south than p2	furtherNorthPoints negation
furtherEastPoints(p1,p2)	Returns true whenever p1 is further east than p2	Using latitudes by considering nodes in west and east hemispheres
furtherWestPoints(p1,p2)	Returns true whenever p1 is further west than p2	furtherEastPoints negation
furtherNorthWays(s1,s2)	Returns true whenever all points of s1 are further north than all points of s2	Using furtherNorthPoints
furtherSouthWays(s1,s2)	Returns true whenever all points of s1 are further south than all points of s2	furtherNorthWays negation
furtherEastWays(s1,s2)	Returns true whenever all points of s1 are further east than all points of s2	Using furtherEastPoints
furtherWestWays(s1,s2)	Returns true whenever all points of s1 are further west than all points of s2	furtherEastWays negation

Figure 4: Coordinate based OSM Operators.

Name	*Definition*	*Clementini's Op.*
inWay(p,s)	Returns true whenever p (point) is in s (way)	Contains
inSameWay(p1,p2)	Returns true whenever p1 (point) and p2 (point) are in the same street	Equals
isCrossing(s1,s2)	Returns true whenever s1 (way) crosses s2 (way)	Crosses
isNotCrossing(s1,s2)	Returns true whenever s1 does not cross s2	Disjoint
isEndingTo(s1,s2)	Returns true whenever s1 ends to s2	Touches (neither initial nor final point)
isContinuationOf(s1,s2)	Returns true whenever s2 is a continuation of s1	Touches (either initial or final point)

Figure 5: Clementini based OSM Operators.

The Clementini based operator `inWay`, which checks whether a point is located in a street, is defined, using Clementini's operator *contains*, as follows:

```
declare function osm:inWay($point as node(), $way as node())
{
  let $point := osm_gml:_osm2GmlPoint($point),
      $line := osm_gml:_osm2GmlLine($way)
  return geo:contains($line,$point)
};
```

The Clementini based operator *inSameWay*, which returns *true* whether two points are located in the same street, uses the auxiliary function *WaysOfaPoint* to retrieve the street (or streets) in which the points are located. *inSameWay* uses the Clementini's operator *equals*, and is defined as follows:

```
declare function osm:inSameWay($node1 as node(), $node2
                          as node(), $document as node()*)
{
  let
      $way1 := osm:WaysOfaPoint($node1,$document),
      $way2 := osm:WaysOfaPoint($node2,$document)
  return
   some $x in $way1 satisfies
   (some $y in $way2 satisfies
    (let $line1 := osm_gml:_osm2GmlLine($x),
             $line2 := osm_gml:_osm2GmlLine($y)
                    return geo:equals($line1,$line2)))
};
```

Now, the Clementini based operator `isCrossing`, which checks if two streets are crossing, is defined, by using Clementini's operator *crosses*, as follows:

```
declare function osm:isCrossing($way1 as node(),
  $way2 as node()) {
  osm:booleanQuery($way1,$way2,"geo:crosses")
};
```

Here, a *Boolean query pattern* is used, called `booleanQuery`, which makes the definition of the Clementini based OSM operators easier, and is defined as follows:

```
declare function osm:booleanQuery($way1 as node(),
  $way2 as node(), $functionName as xs:string)
{
  let $mutliLineString1 := osm_gml:_osm2GmlLine($way1),
      $multiLineString2 := osm_gml:_osm2GmlLine($way2)
  let $spatialFunction :=
                 fn:function-lookup(xs:QName($functionName),2)
  return $spatialFunction($mutliLineString1,$multiLineString2)
};
```

This pattern takes as parameters two *streets* and a *functionName*. *functionName* is a Clementini's operator from JTS, applied to the above streets. The Boolean query pattern is also used for the implementation of `isNotCrossing`. The cases `isEndingTo` and `isContinuationOf` are special cases of OSM operators that are not direct instances of the Boolean query pattern. They can be derived from Clementini's spatial operators. These functions use Clementini's operator *touches*, as well as start and end point of the street in order to check the ending or continuation of the street. For instance, `isEndingTo` is defined as follows:

```
declare function osm:isEndingTo($way1 as node(),
                               $way2 as node())
{
 if (osm:booleanQuery($way1,$way2,"geo:touches"))
 then
```

Name	*Semantics*
fn:for-each(s,f)	Applies the function f to every element of the sequence s
fn:filter(s,p)	Selects the elements of the sequence s for which p is true
fn:for-each-pair(s1,s2,f)	Zips the elements of s1 and s2 with the function f
fn:fold-left(s,e,f)	Folds (left) the sequence s with f starting from e
fn:fold-right(s,e,f)	Folds (right) the sequence s with f starting from e

Figure 6: Higher order functions of XQuery.

```
    let $mutliLineString1 := osm_gml:_osm2GmlLine($way1),
        $multiLineString2 := osm_gml:_osm2GmlLine($way2),
        $intersection_point :=
        geo:intersection($mutliLineString1,$multiLineString2),
        $start_point := geo:start-point($mutliLineString1/*),
        $end_point := geo:end-point($mutliLineString1/*)
    return
          (geo:equals($intersection_point/*,$start_point/*) or
          geo:equals($intersection_point/*,$end_point/*))
    else false()
};
```

3.4 Higher Order XQuery Facilities

XQuery 3.0 is equipped with higher order facilities. Basically, XQuery provides a library of higher order functions, that is, a repertoire of functions having themselves functions as arguments. It is possible due to the use of XQuery type `function(item()) as item()*`. Figure 6 shows the set of higher order functions available in XQuery, which adds new functionality to our library in a double sense:

(a) Allowing query composition by combining higher order functions and OSM operators, and

(b) Allowing keyword based search queries by combining higher order functions and keyword based search operators

For instance, with respect to (a), the higher order function `filter` combined with the OSM spatial operator `isCrossing` can be used, in order to get all the streets crossing a given street (for instance, *"Calzada de Castro"* street in Almería city) as follows. Let us remark the natural interpretation and simplicity of this query.

```
fn:filter(rt:getLayerByName(.,"Calle Calzada de Castro"),
osm:isCrossing(?, osm:getOneWay(., "Calle Calzada de Castro")))
```

Here, `getLayerByName` obtains all the streets close to *"Calle Calzada de Castro"* street[8] from the indexed OSM layer, and `getOneWay` retrieves *"Calzada de Castro"* street (i.e., the OSM way representing *"Calzada de Castro"*). The symbol "?" indicates the `isCrossing` argument to be filtered.

[8]"Calle" means street in spanish.

With respect to (b), a new repertoire of functions has been defined for adding (`addTag`), removing (`removeTag`), replacing (`replaceTag`) and retrieving (`searchOneTag` and `searchTags`) *keywords* of OSM maps. For instance, the function `searchTags`, which searches a set of keywords in a way, is defined as follows:

```
declare function osm:searchTags($node as node(),
   $collectionValueToSearch as xs:string*)
{
  some $value in
  (distinct-values(
  for $valueToSearch in $collectionValueToSearch
    return osm:searchOneTag($node,$valueToSearch)))
        satisfies ($value = true()))
};
```

For instance, `searchTags` in combination with the higher order function `filter` can be used to retrieve all the schools close to *"Calzada de Castro"* street from the indexed OSM map. Let us highlight that we work with geo-localized queries; i.e., keyword search is restricted to a geo-localized point or street. In this case the search is restricted to *"Calzada de Castro"* street.

```
fn:filter(rt:getLayerByName(.,"Calle Calzada de Castro"),
osm:searchTags(?,"school"))
```

4 EXAMPLES

In this section, we will show some examples of the use of our library in order to query OSM map data. In addition, we will also provide benchmarks from datasets of several sizes. Assuming the map of Figure 1 (i.e. Almería city), we can consider the following batch of queries whose results are shown in Figure 7.

Example 1. Retrieve the schools and high schools close to *"Calzada de Castro"* street:

```
fn:filter(
rt:getLayerByName(.,"Calle Calzada de Castro"),
osm:searchTags(?,("high school", "school")))
```

In this query, the higher order function `filter` in combination of the function `searchTags` is used. It enables the retrieval of the schools and high schools from the layer; i.e. to search for the keywords *school* and *high school* from the tags included in the layer objects[9]. The R-tree has been previously loaded in memory of the XQuery interpreter, and the function `getLayerByName` retrieves from the R-tree, the nodes

[9]Although here we cannot work with spatial operators for buildings, we are still able to formulate keyword based search queries.

Figure 7: Results for Examples.

and ways close to *"Calzada de Castro"* street (i.e., those objects whose MBR's overlap with the MBR of *"Calzada de Castro"* street).

Example 2. Retrieve the streets crossing *"Calzada de Castro"* and ending to *"Avenida Montserrat"* street:

```
let $waysCrossing :=
fn:filter(
rt:getLayerByName(.,"Calle Calzada de Castro"),
osm:isCrossing(?, osm:getOneWay(., "Calle Calzada de Castro")))
return
fn:filter($waysCrossing,
osm:isEndingTo(?, osm:getOneWay(., "Avenida Montserrat")))
```

Here, the function `filter` has been used in combination with the OSM operators `isCrossing` and `isEndingTo`. In this query, first of all the streets crossing *"Calzada de Castro"* street are filtered, and then, from these streets, the streets ending to *"Avenida de Montserrat"* street are filtered.

Example 3. Retrieve the schools close to a street, wherein *"Calzada de Castro"* street ends.

```
let $waysAllEndingTo :=
fn:filter(
rt:getLayerByName(.,"Calle Calzada de Castro"),
osm:isEndingTo(osm:getOneWay(., "Calle Calzada de Castro"),?))
return
fn:filter(
fn:for-each($waysAllEndingTo, rt:getLayerByOneWay(.,?)),
osm:searchTags(?,"school"))
```

Here, we can see how both kinds of queries can be combined: on the one hand, the OSM operator `isEndingTo` is used to get the streets wherein *"Calzada de Castro"* street ends, and, on the other hand, the keyword *school* from the nodes occurring in the layer of each street is searched. `filter` is used twice.

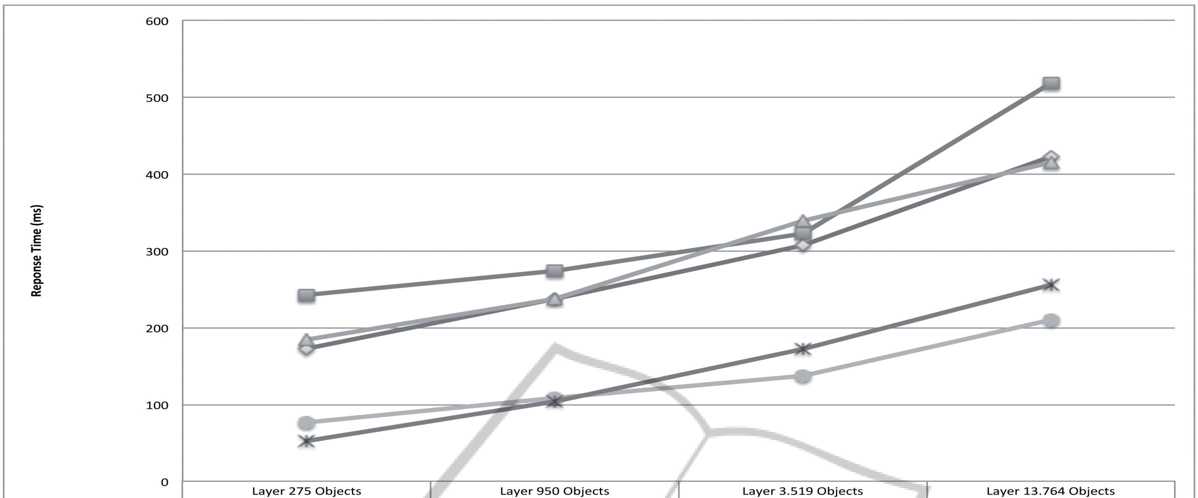
Example 4. Retrieve the streets close to *"Calzada de Castro"* street, in which there is a supermarket *"El Arbol"* and a pharmacy (or chemist's).

```
osm:intersectionQuery(
 osm:unionQuery(
 rt:getLayerByName(.,"El Arbol"),
 rt:getLayerByName(.,"pharmacy")),
 rt:getLayerByName(.,"Calle Calzada de Castro"))
```

Here, we can see an additional feature of our library; i.e. *the handling of set-based operators*, such as `union`, `intersection` and `difference` of sequences. Functions `unionQuery`, `intersectionQuery` and `exceptQuery` of the library can be used to produce more complex queries. In this case, the intersection of streets close to *"Calzada de Castro"* street with a supermarket (named *"El Arbol"*) and a pharmacy is requested.

Example 5. Retrieve the streets to the north of *"Calzada de Castro"* street:

```
fn:filter(
```



	Layer 275 Objects	Layer 950 Objects	Layer 3.519 Objects	Layer 13.764 Objects
Example Query 1	75,92	108,45	137,89	209,45
Example Query 2	242,54	273,23	322,77	518,08
Example Query 3	172,59	236,63	306,64	422,67
Example Query 4	52,42	103,87	172,24	255,58
Example Query 5	184,29	237,06	339,62	414,86

Figure 8: Benchmarking of examples for datasets of different sizes.


```
rt:getLayerByName(.,"Calle Calzada de Castro"),
osm:furtherNorthWays(
osm:getOneWay(., "Calle Calzada de Castro"),?))
```

Finally, we can see the use of geo-positioning queries. Streets close to *"Calzada de Castro"* street are obtained, and then, the further north streets are filtered.

4.1 Benchmarks

Now we would like to show the benchmarks obtained from the previous examples, for datasets of different sizes. We have used the *BaseX Query* processor in a Mac Core 2 Duo 2.4 GHz. All benchmarking proofs have been tested using a virtual machine running Windows 7 since the *JTS Topology Suite* is not available for *Mac OS BaseX* version. Benchmarks are shown in milliseconds in Figure 8.

We have tested Examples 1 to 5 with sizes ranging from two hundred to fourteen thousand objects, corresponding to: from a zoom to *"Calzada de Castro"* street to the whole Almería city map (around 10 square kilometers). From the benchmarks, we can conclude that increasing the map size, does not increase, in a remarkable way, the answer time.

Unfortunately, we cannot compare our benchmarks with existent implementations of similar tools due to the following reasons. Even when OSM has been used for providing benchmarks in a recent work (Eiter et al., 2014), they use OSM as dataset for Description Logic based reasoners rather than to evaluate spatial queries. There are some proposals for defining spatial datasets for benchmarking Spatial RDF stores (Kolas, 2008; Garbis et al., 2013), mainly focused on Clementini's and Egenhofer's operators whereas our query language offers more sophisticated queries.

5 RELATED WORK

GQuery (Boucelma and Colonna, 2004) is a proposal for adding spatial operators to *XQuery*. Manipulation of trees and sub-trees are carried out by *XQuery*, while spatial processing is performed using geometric functions and *JTS*. *GeoXQuery* approach (Huang et al., 2009) extends the *Saxon* XQuery processor (Kay, 2008) with function libraries that provide geo-spatial operations. It is also based on *JTS* and provides a GML to SVG transformation library for the XQuery processor in order to show query results. *GML Query* (Li et al., 2004) is also a contribution in this research line that stores GML documents in a spatial RDBMS. This approach performs a simplification of the GML schema that is then mapped to its corresponding relational schema. The basic values of spatial objects are stored as values of the tables. Once the document is stored, spatial queries can be expressed using the *XQuery* language with spatial functions. The queries are translated to their equivalent in SQL which are executed by means of the spatial RDBMS.

Linked Geospatial Data is an emerging line of research (see (Koubarakis et al., 2012) for a survey) focused on the handling of RDF based representation of geo-spatial information, adopting a Semantic Web point of view (Egenhofer, 2002), and using SPARQL style query languages like SPARQL-ST (Perry et al., 2011), stSPARQL (Koubarakis and Kyzirakos, 2010) and GeoSPARQL (Battle and Kolas, 2012). The *LinkedGeoData* dataset (Stadler et al., 2012) is a work of the AKSW research group at the University of Leipzig that uses GeoSPARQL and well-known text (WKT) RDF vocabularies to represent OSM data.

In our approach, we have followed the same direction as (Boucelma and Colonna, 2004; Huang et al., 2009; Li et al., 2004), adopting XQuery for querying, but they are not focused on OSM, and higher order functions are not used. With regard to *OSM3S* (i.e., *Overpass API*), it is specifically designed for search criteria like location, types of objects, tag values, proximity or combinations of them. *Overpass API* has the query languages Overpass XML and Overpass QL. Both languages are equivalent. They handle OSM objects ((a) standalone queries) and set of OSM objects ((b) query composition and filtering). With respect to (a), the query language allows the expression of queries in order to search a particular object, and is equipped with forward or backward recursion to retrieve links from an object (for instance, it allows to retrieve the nodes of a way). With respect to (b), the query language allows the expression of queries using several search criteria. Among others, it can express: to find all data in a bounding box (i.e., positioning), to find all data near something else (i.e., proximity), to find all data by tag value (exact value, non-exact value and regular expressions), negation, union, difference, intersection, and filtering, with a rich set of selectors, and by polygon, by area pivot, and so on. However, *Overpass API* facilities (i.e., query composition and filtering) cannot be combined with spatial operators such as Clementini's crossing or touching. In *Overpass API*, only one type of spatial intersection is considered (proximity 0 by using across selector). For instance, the query (allowed in our library) *"Retrieve the streets crossing Calzada de Castro street and ending to Avenida de Montserrat street"* is not allowed in *Overpass API*. On the other hand, *Overpass API* has a rich query language for keyword search based queries. We plan to extend our library to handle a richer set of keyword search based queries.

SPARQL based query languages offer a rich set of spatial operators. For instance *stSPARQL* (Koubarakis and Kyzirakos, 2010) is equipped with Clementini's operators, as well as MBRs based operators. Also directional operators are considered and, functions for constructing new objects are included: buffer, boundary, envelope, convexHull, union, intersection and difference as well as distance-based operators: distance and area. Finally, temporal operators are also considered. The RDF representation of OSM and the use of SPARQL style query languages, offer also the opportunity to describe more complex queries than OSM3S and XAPI. The use of XQuery for OSM data makes sense when the XML representation of an OSM layer is the input of a query, and the answer is also required in XML format; for instance, when using JOSM to visualize OSM maps. Unfortunately, *SPARQL* and its spatial dialects are not equipped with Higher Order (although there exists a recent proposal (Atzori, 2014) for SPARQL) and thus, the queries that we can propose, are impossible to express in them. Even more, spatial dialects of *SPARQL* have to deal with the graph based structure of OSM RDF, that sometimes can make it more difficult, if not impossible, the expression of some queries (Alkhateeb et al., 2011). The same can happen when using a spatial DBMS and OSM data are imported to it. While spatial DBMS can offer the same functionality than the proposed XQuery extension, higher order facilities makes the work easier.

6 CONCLUSIONS AND FUTURE WORK

We have presented an XQuery library for querying OSM. We have defined a set of OSM Operators suitable for querying points and streets from OSM. We have shown how higher order facilities of XQuery enable the definition of complex queries over OSM involving composition and keyword searching. We have provided some benchmarks using our library that take profit from the R-tree structure used to index OSM. As future work firstly, we would like to extend our library to handle closed ways of OSM, in order to query about buildings, parks, etc. Secondly, we would like to enrich the repertoire of OSM operators for points and streets: distance based queries, ranked queries, etc. Finally, we would like to develop a JOSM plugin, as well as a Web site, with the aim to execute and to show results of queries directly in OSM maps.

ACKNOWLEDGEMENTS

This work was funded by the EU ERDF and the Spanish Ministry of Economy and Competitiveness (MINECO) under Project TIN2013-44742-C4-4-R, and by the Andalusian Regional Government (Spain) under Project P10-TIC-6114.

REFERENCES

Alkhateeb, F., Baget, J.-F., and Euzenat, J. (2011). Extending SPARQL with regular expression patterns (for querying RDF). *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(2):57–73.

Atzori, M. (2014). Toward the Web of Functions: Interoperable Higher-Order Functions in SPARQL. In *The Semantic Web–ISWC 2014*, pages 406–421. Springer.

Bamford, R., Borkar, V., Brantner, M., Fischer, P. M., Florescu, D., Graf, D., Kossmann, D., Kraska, T., Muresan, D., Nasoi, S., et al. (2009). XQuery reloaded. *Proceedings of the VLDB Endowment*, 2(2):1342–1353.

Battle, R. and Kolas, D. (2012). Enabling the geospatial semantic web with Parliament and GeoSPARQL. *Semantic Web*, 3(4):355–370.

Berglund, A., Boag, S., Chamberlin, D., Fernandez, M., Kay, M., Robie, J., and Siméon, J. (2010). XML path language (XPath) 2.0. *W3C*.

Boucelma, O. and Colonna, F. (2004). GQuery: a Query Language for GML. In *Proc. of the 24th Urban Data Management Symposium*, pages 27–29.

Clementini, E. and Di Felice, P. (2000). Spatial operators. *ACM SIGMOD Record*, 29(3):31–38.

Egenhofer, M. J. (1994). Spatial SQL: A Query and Presentation Language. *IEEE Trans. Knowl. Data Eng.*, 6(1):86–95.

Egenhofer, M. J. (2002). Toward the semantic geospatial web. In *Proceedings of the 10th ACM international symposium on Advances in geographic information systems*, pages 1–4. ACM.

Eiter, T., Schneider, P., Šimkus, M., and Xiao, G. (2014). Using OpenStreetMap Data to Create Benchmarks for Description Logic Reasoners. In *Proceedings of the 3rd International Workshop on OWL Reasoner Evaluation (ORE 2014)*, pages 51–57. CEUR Workshop Proceedings, Vol-1207.

Garbis, G., Kyzirakos, K., and Koubarakis, M. (2013). Geographica: A Benchmark for Geospatial RDF Stores. In *The Semantic Web–ISWC 2013*, pages 343–359. Springer.

Grun, C. (2015). BaseX. The XML Database. http:// basex.org.

Hadjieleftheriou, M., Manolopoulos, Y., Theodoridis, Y., and Tsotras, V. J. (2008). R-Trees–A Dynamic Index Structure for Spatial Searching. In *Encyclopedia of GIS*, pages 993–1002. Springer.

Haklay, M. and Weber, P. (2008). Openstreetmap: User-generated street maps. *Pervasive Computing, IEEE*, 7(4):12–18.

Huang, C.-H., Chuang, T.-R., Deng, D.-P., and Lee, H.-M. (2009). Building GML-native web-based geographic information systems. *Computers & Geosciences*, 35(9):1802–1816.

Kay, M. (2008). Ten reasons why saxon xquery is fast. *IEEE Data Eng. Bull.*, 31(4):65–74.

Kolas, D. (2008). A Benchmark for Spatial Semantic Web Systems. In *International Workshop on Scalable Semantic Web Knowledge Base Systems*.

Koubarakis, M., Karpathiotakis, M., Kyzirakos, K., Nikolaou, C., and Sioutis, M. (2012). Data Models and Query Languages for Linked Geospatial Data. In *Reasoning Web. Semantic Technologies for Advanced Query Answering*, pages 290–328. Springer.

Koubarakis, M. and Kyzirakos, K. (2010). Modeling and querying metadata in the semantic sensor web: The model stRDF and the query language stSPARQL. In *The semantic web: research and applications*, pages 425–439. Springer.

Li, Y., Li, J., and Zhou, S. (2004). GML Storage: A Spatial Database Approach. In Wang, S., Yang, D., Tanaka, K., Grandi, F., Zhou, S., Mangina, E. E., Ling, T. W., Song, I.-Y., Guan, J., and Mayr, H. C., editors, *ER (Workshops)*, volume 3289 of *Lecture Notes in Computer Science*, pages 55–66. Springer.

Meier, W. (2003). eXist: An open source native XML database. In *Web, Web-Services, and Database Systems*, pages 169–183. Springer.

Perry, M., Jain, P., and Sheth, A. P. (2011). SPARQL-ST: Extending SPARQL to support spatiotemporal queries. In *Geospatial semantics and the semantic web*, pages 61–86. Springer.

Robie, J., Chamberlin, D., Dyck, M., and Snelson, J. (2014). XQuery 3.0: An XML query language. *W3C*.

Shekhar, S. and Xiong, H. (2008). Java topology suite (jts). In *Encyclopedia of GIS*, pages 601–601. Springer.

Stadler, C., Lehmann, J., Höffner, K., and Auer, S. (2012). Linkedgeodata: A core for a web of spatial open data. *Semantic Web*, 3(4):333–354.