Getting practical with GeoSPARQL and Apache Jena

Simon Bin, Claus Stadler[1], Lorenz Bühmann and Michael Martin[2]

Institute for Applied Informatics (InfAI), Leipzig, Germany
[1]*Leipzig University, Leipzig, Germany*
[2]*Chemnitz University of Technology, Chemnitz, Germany*

Abstract

This paper explores the integration of geo-spatial data into RDF (Resource Description Framework) using Apache Jena, a popular Java-based framework for building Semantic Web applications. We explain the basic representation of geo-spatial data in RDF with a focus on both the new GeoSPARQL 1.1 standard and Apache Jena. Our investigation covers advanced techniques, such as transformation of coordinate reference systems, aggregation of geo-spatial data, creation of new geo-objects, and simplification of polygons. Additionally, we discuss the usage of the H3 Grid as a Discrete Global Grid System (DGGS) for geo-spatial conversion. Furthermore, we present performance optimisations specific to Apache Jena, including per-graph geo-indexing, improved geo-index serialization for faster startup times, and manual optimisation of geo-spatial queries. We conclude with a comparison of different geo-functions and outline future directions for enhancing geo-spatial data management in RDF.

Keywords

GeoSPARQL, Free software, Apache Jena, RDF, spatial, geometry, GIS

1. Introduction

Semantic Web technologies are an enabler for building harmonised views over data assets in the form of a knowledge graph (KG). Key aspects of the Semantic Web include RDF as a uniform data model, vocabularies as a means to model data in a decentralised way, and yet facilitate easy integration by conformance to standards, community conventions, or best practices. Ontology languages – such as OWL – enable adding detailed specifications to conceptualisations.

However, besides data modelling aspects and reasoning, different domains introduce their own set of data objects with corresponding operations. In practice, a mere RDF view of cross-domain data is insufficient, because there is also the need for a KG system to support storage, indexing, and operations of relevant domain-specific data types.

GeoSPARQL is an extension that builds upon RDF and SPARQL. Spatial data conventionally falls into the categories of vector and raster data. In the domain of supply chain information, spatial data plays a crucial role. In this work, we study a selection of supply chain-related use cases which we implemented on an Open Source stack based on Apache Jena.[1] We present our findings on which features are still missing in GeoSPARQL itself, and which parts need an improved implementation. We also provide implementations of our own extensions as a basis for further discussion.

2. Related work

The importance of using geo-spatial data in a knowledge graph goes back for more than a decade, such as the early efforts to lift OpenStreetMap data into knowledge graphs [1]. Nowadays, OpenStreetMap (OSM) data can be quickly converted to RDF using the osm2rdf tool [2]. (Originally, the code was not following the standard, thus impairing reuse. We raised an issue about this[2] and now it can be

6th International Workshop on Geospatial Linked Data (GeoLD2024) at ESWC 2024, May 26th or May 27th, 2024, Hersonissos, Greece
sbin@informatik.uni-leipzig.de (S. Bin)
0000-0001-9948-6458 (C. Stadler)

[1]https://jena.apache.org
[2]https://github.com/ad-freiburg/osm2rdf/issues/24

used with any compliant GeoSPARQL store successfully.) A recent summary on knowledge graph construction [3] also emphasises the ongoing interest. The community, such as Knowledge Graphs in Action and EuroSDR [4], is also discussing this topic.

The potential benefit of connecting geo-spatial data with knowledge graph entities such as from Wikidata has been noted in [5], too. Furthermore, knowledge graphs can be interlinked based on their geo-spatial data, as shown in several approaches such as [6] or [7]. Modelling the data according to well-defined standards is important for these usages as well as for the FAIR data principles [8]. With regards to interoperability, GeoSPARQL-conformance provides a foundation for the realisation of advanced use cases in a modular and vendor-independent way. One such example is geo-spatial question answering over knowledge graphs using GeoSPARQL queries, as proposed in [9].

3. Geo-spatial data in RDF

In this section, we describe how to model geo-spatial data using RDF. In the following use cases, we contextualise the (required) standard according to the requirements and the current state with regards to Apache Jena GeoSPARQL [10]. Apache Jena achieved the highest compliance score in the GeoSPARQL Benchmark [11]. GeoSPARQL 1.1 [12] was recently released (2024-01-29) and is a great update to the version 1.0 standard, however, it has not yet been fully implemented in Apache Jena.

3.1. Storing and querying geo-spatial data using RDF and (Geo)SPARQL

The first requirement of working with geo-spatial data should be to store it in the knowledge graph. Initial attempts at this were to encode the latitude and longitude of an object on the earth using the wgs84 vocabulary.[3] The modern way is to use GeoSPARQL with a sub-language for geometries, this has the advantage of supporting lines and (multi-)polygons in addition to points. The most popular choice here is the *Well-known text* (WKT) representation [13] (see Figure 1), but other representations are possible, such as the Geography Markup Language (GML).

```
## GeoSPARQL standard
PREFIX geo:      <http://www.opengis.net/ont/geosparql#>
PREFIX geof:     <http://www.opengis.net/def/function/geosparql/>
## Jena Spatial vendor extensions
PREFIX spatial:  <http://jena.apache.org/spatial#>
PREFIX spatialF: <http://jena.apache.org/function/spatial#>
## JenaX web service extension
PREFIX url:      <http://jsa.aksw.org/fn/url/>
## osm2rdf OpenStreetMap vocabulary
PREFIX osmkey:   <https://www.openstreetmap.org/wiki/Key:>
PREFIX osmrel:   <https://www.openstreetmap.org/relation/>
PREFIX osmway:   <https://www.openstreetmap.org/way/>
PREFIX osm:      <https://www.openstreetmap.org/>
## CoyPu project vocabulary
PREFIX coy:      <https://schema.coypu.org/global#>
## MoIn project vocabulary
PREFIX moino:    <http://mobilityindex.net/ontology/>
```

Used prefixes in this document

```
<MyTriangle> a geo:Feature ;
  geo:hasGeometry [ a geo:Geometry ;
    geo:asWKT
        "POLYGON((26.5 45, 27 45, 27 45.2, 26.5 45))"^^geo:wktLiteral
  ] ;
  geo:hasCentroid [ a geo:Geometry ;
    geo:asWKT "POINT(26.83 45.07)"^^geo:wktLiteral ]
```

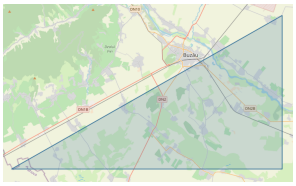

Figure 1: Representation of a feature with a geometry and a centroid point using GeoSPARQL/RDF

[3] https://www.w3.org/2003/01/geo/

This is possible with **GeoSPARQL 1.0** (before 2022, there was an error in the specification that prevented the use of conventional GML in conforming implementations such as the Apache Jena version 4.6.0 [14] – this might suggest that GML is not being used widely in the GeoSPARQL/RDF communities). **GeoSPARQL 1.1** adds support for even more representation languages, such as GeoJSON.

After the information is represented like this, the whole range of geo-spatial functions provided by GeoSPARQL can be used in SPARQL queries. Commonly used functions include `geo:sfWithin`/`geof:sfWithin`, `geo:sfIntersects`/`geof:sfIntersects`, etc., and can be used in linking or querying to correlate geo-spatial objects, like `?company geo:sfWithin ?country` (the geometry of a company is geographically located within the polygon of a country) or `FILTER(geof:sfIntersects(?riverGeom, ?regionGeom)` (only include those SPARQL results where the geometry of the river intersects the geometry of a specific region).

Additionally, there can be more than one geometry information available for an entity, for example, the polygon describing the extent of the object as well as the centre point (centroid). There are standardised properties for this information available starting with **GeoSPARQL 1.1**.

Finally, it is possible to do mathematical computations on geo-spatial objects. The function `geof:distance` (**GeoSPARQL 1.0**) can be used to calculate the distance between two objects, as well as the vendor function `spatialF:distance` in Apache Jena and `geof:metricDistance` in **GeoSPARQL 1.1**. In GeoSPARQL 1.1 it is also possible to calculate the area of an object with the functions `geof:area` or `geof:metricArea`.

3.2. Transformation of Coordinate Reference System

Oftentimes, especially when integrating data from different sources, the geo-spatial data is not in the same Coordinate Reference System (CRS). For example, the same polygon as above, when expressed in the CRS with EPSG[4] code 3844[5] looks like this:

```
"<http://www.opengis.net/def/crs/EPSG/0/3844> POLYGON((390018.693878 618366.368293,
↪  390877.440409 657778.063276, 413096.430405 657224.263781, 390018.693878
↪  618366.368293))"^^geo:wktLiteral
```

Recording the CRS in the data is supported in **GeoSPARQL 1.0** and is done by adding a CRS IRI in front of the WKT literal. It is the most practical to work with integrated data if it is all expressed in the same CRS. For that, CRS transformations are required.

This possibility to transform the CRS exists in Apache Jena with a custom vendor function: `spatialF:transformSRS(?originalGeometry, <http://www.opengis.net/def/crs/EPSG/0/3844>)`. It has been standardised in **GeoSPARQL 1.1** as `geof:transform(?geom, ?crsIRI)`.

Support for CRS transformations is currently crucial in Apache Jena to calculate metric buffers and distances. (Buffering can be used for many purposes, such as finding objects within a certain distance to the boundary or intersection checks with an error margin.) Consequently, in order to calculate a metric buffer around a polygon, the following SPARQL query is required:

```
BIND(geof:transform(?geo, <http://www.opengis.net/def/crs/EPSG/0/3844>) AS ?metricGeo)
BIND(geof:buffer(?metricGeo, 11113.9, uom:metre) AS ?metricBuffered)
BIND(geof:transform(?metricBuffered, <http://www.opengis.net/def/crs/OGC/1.3/CRS84>) AS
↪  ?geoBufferedCrs84)
```

Otherwise, it is possible to calculate the buffer by adding degrees of longitude/latitude, however, this has a distorting effect due to the shape of the earth, as can be seen in Figure 2. We have opened a clarification request as GeoSPARQL issue 398.[6]

[4] European Petroleum Survey Group
[5] https://epsg.io/3844
[6] https://github.com/opengeospatial/ogc-geosparql/issues/398

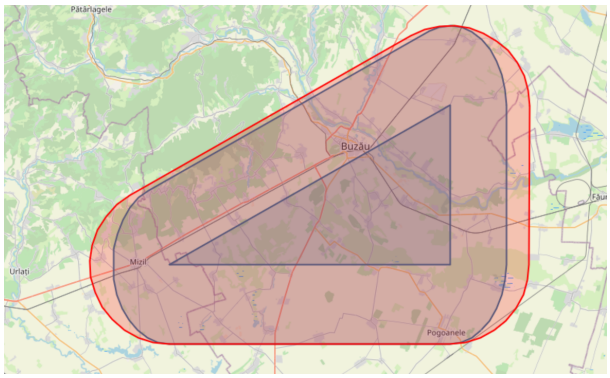

Figure 2: Difference between metric buffering (red) and buffering by degree longitude and latitude in Apache Jena

3.3. Transformation of representation language

In addition to the CRS, it can be desirable to transform between representation language of the geometries. On the one hand, processing of integrated data is easier when it is all using the same representation (we found WKT to be predominant). One use case is the consumption of disaster data from the Global Disaster Alert and Coordination System (GDACS). GeoJSON data is provided for various disaster events, for example, the tropical cyclone number 1001053,[7] as shown in Figure 3. On the other hand, some other applications such as the map viewer Leaflet can also work more natively with GeoJSON.

Starting with **GeoSPARQL 1.1**, conversion functions between representations have been added. They are using the `geof:as***` naming convention: `geof:asGeoJSON`, `geof:asGML`, and `geof:asWKT`. We have already implemented the required support in our **JenaX extension** [14], but the function names still need to be aligned with GeoSPARQL 1.1.

```
VALUES ?geojsonUrl {
  <https://www.gdacs.org/contentdata/resources/TC/1001053/
  ↪  geojson_1001053_2.geojson>
}
BIND(strdt(url:text(?geojsonUrl), geo:geoJSONLiteral) AS ?geojson)
## currently implemented:
BIND(spatialF:transformDatatype(?geojson, geo:wktLiteral) AS ?wktGeom)
## standards compliant:
BIND(geof:asWKT(?geojson) AS ?wktGeom)
```

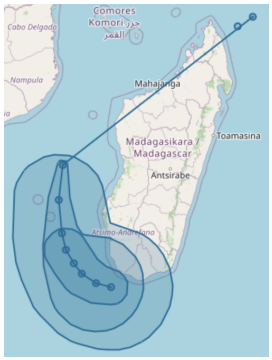

Figure 3: Consumption of disaster data from GDACS and conversion to WKT (vendor functions)

3.4. Aggregating geo-spatial data

When assembling open geo-spatial data, the OpenStreetMap[8] project is a huge treasure trove of collaboratively collected map data of the world. However, many geo-spatial objects in OSM are modelled as consisting of many parts, like many piece-wise relations of a river or a railroad network.

With GeoSPARQL 1.1, geo-spatial aggregate functions were introduced that can be used in the SPARQL queries together with the `GROUP BY` clause. **GeoSPARQL 1.1** defines the functions `geof:aggBoundingBox`, `geof:aggBoundingCircle`, `geof:aggCentroid`, `geof:aggConcaveHull`, `geof:aggConvexHull`, and `geof:aggUnion`.

[7] https://gdacs.org/report.aspx?eventid=1001053&eventtype=TC
[8] https://www.openstreetmap.org/about

```
SELECT
      ## currently implemented:
      (geof:lineMerge(geof:collect(?wayGeom)) AS ?riverGeom)
      ## possible standards suggestion?
      (geof:aggConcatLines(?wayGeom) AS ?riverGeom)
  WHERE {
    ?s a osm:relation ;
       osmkey:name "Elbe" ;
       osmrel:member/osm:id ?m .
    ?m geo:hasGeometry/geo:asWKT ?wayGeom .
}
```

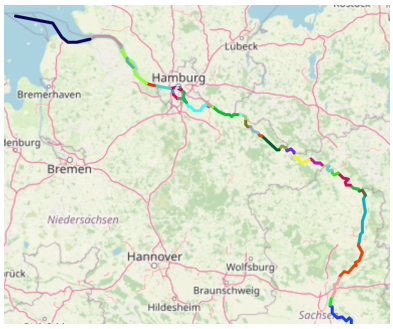

Figure 4: Example SPARQL query to reconstruct the Elbe river using aggregate functions

However, for our use case, we would also need a function to aggregate line strings. Such a function was proposed in the GeoSPARQL 1.1 Motivations paper [15] but it is absent from the released standard. We have opened a request for inclusion in a later version as GeoSPARQL issue 402.[9] Furthermore, a function `geof:aggCollect` to create geometry collections of geometries has been suggested for inclusion in a later version of GeoSPARQL. These functions are already available in our **JenaX** extension.

Such aggregate functions can then be used to combine the segments of a river back into a single river object, which is oftentimes more practical for our use case than having to deal with the individual pieces. This is shown in Figure 4.

3.5. Creating new geo-objects

It can be useful to create a connection between multiple points through a line, this also makes it easier to detect which areas are crossed by a path connecting multiple points. To assist with this, we provide the `geof:makeLine` function that turns a geometry collection into a line. Similarly, it can be useful to create new points, especially with longitude and latitude at hand, that may be coming from legacy properties. This can be done using the `spatialF:convertLatLon` function (the function name also makes it hard to mix up latitude and longitude in the parameters). The query in Figure 5 demonstrates how to achieve a visualisation of a company's suppliers based on their coordinates. It results in line strings from Frankfurt, Germany, to a list of companies that are geo-coded using legacy latitude and longitude properties from the *CoyPu* project.[10] These functions are implemented in our **JenaX** extension.

```
SELECT
      (geof:makeLine(geof:union(?frankfurtGeom, ?supplierGeom)) AS ?lineGeom)
  WHERE {
    GRAPH <https://data.coypu.org/cities/> {
      <http://www.wikidata.org/entity/Q1794> geo:hasGeometry/geo:asWKT
      ↪  ?frankfurtGeom
    }
    GRAPH <https://data.coypu.org/companies/demo/> {
      ?s coy:isSupplierOf "ferrero" ;
         coy:hasLatitude ?lat ;
         coy:hasLongitude ?lon .
      BIND(spatialF:transformSRS(spatialF:convertLatLon(?lat, ?lon),
      ↪  <http://www.opengis.net/def/crs/OGC/1.3/CRS84>) AS ?supplierGeom)
    }
}
```

Figure 5: Air lines from Frankfurt, Germany to some suppliers in the *CoyPu* project knowledge graph

[9] https://github.com/opengeospatial/ogc-geosparql/issues/402
[10] https://coypu.org/

3.6. Simplification of polygons

Some polygons, especially those of administrative boundaries, can have very detailed borders whose size in WKT representation amounts to dozens of megabytes. It can be useful to simplify such geospatial objects, either to improve the performance of map visualisation, to speed up containment checks when accuracy is less important, or to reduce the size requirements of the polygon. We are providing a function `geof:simplifyDp` or `geof:simplifyVw` that uses the Douglas-Peucker algorithm or the Visvalingam-Whyatt algorithm to simplify polygons.

These functions are implemented in our **JenaX** extension and have also been suggested by others on the GeoSPARQL issue 257.[11] Such a function has also been suggested for inclusion in a later version of GeoSPARQL.

3.7. Usage of H3 Grid as DGGS

Discrete global grid systems (DGGS) that sub-divide the earth into a grid have gained importance, as can be seen from [16]. As such, it is now (in its abstract form) also part of **GeoSPARQL 1.1**. DGGS can be used to speed up searching for nearby objects located in the same cell. One previous work reports on attempting the AusPIX DGGS [17]. Another popular DGGS used by the company Uber is called H3.[12] We have implemented functions for working with H3 in **JenaX**. The currently implemented functions are: `geof:h3CellIdToGeom`, `geof:h3CellIdToParent`, `geof:h3CellResolution`, `geof:h3GridDistance`, `geof:h3IsValidCell`, `geof:h3LongLatAsCellId`, `geo:h3_cellIdToChildren`, `geo:h3_geometryToCellIds`, and `geo:h3_gridDisk`.

One example of working with H3 in our current extension is shown in Figure 6. In the example, we are tracing existing public transport routes from the *MoIn* project[13] using the H3 grid cells that contain them. This serves as a basis for analysing public transport within a region, such that reachability can be assessed on cell granularity. The next step is to align the H3 functions with GeoSPARQL 1.1. This process might identify missing functionalities for working with DGGS in the current GeoSPARQL standard.

```
SELECT ?routeLineString ?cellGeom {
  [] moino:route ?routeLineString .
  ?cellId geo:h3_geometryToCellIds(?routeLineString 5
  ↪  true)
  BIND(geof:h3CellIdToGeom(?cellId) AS ?cellGeom)
}
```

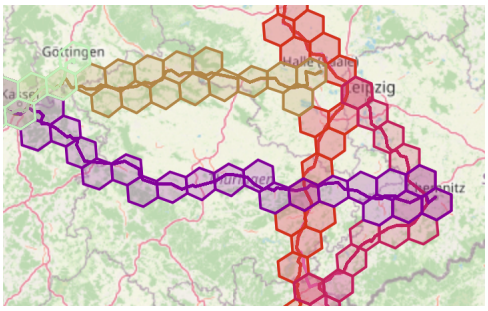

Figure 6: Covering public transport routes in Germany with H3 grid cells.

3.8. Longitude and Latitude of a point

Latitude and longitude are a source of frequent sorrow,[14] because it does not seem to be clear which one is the first coordinate and which one is the second. This confusion is further emphasised by the two CRS CRS:84[15] and EPSG:4326[16] that are both in widespread use. Both refer to the World Geodetic System 1984 (WGS 84), but CRS:84 uses the coordinate order (longitude, latitude), whereas EPSG code 4326 uses the (latitude, longitude) order. CRS:84 is also the default CRS for GeoSPARQL WKT literals.

[11] https://github.com/opengeospatial/ogc-geosparql/issues/257
[12] https://h3geo.org/
[13] https://mobilityindex.net/
[14] https://postgis.net/documentation/tips/lon-lat-or-lat-lon/
[15] http://www.opengis.net/def/crs/OGC/1.3/CRS84
[16] http://www.opengis.net/def/crs/EPSG/0/4326

In our project, we faced raw input data where the coordinates were mixed up for some records only. Even today, the Geo-3D view on the popular SPARQL web interface Yasgui[17] incorrectly plots points on the earth with latitude and longitude reversed. For this reason, we are suggesting to have explicit functions `geof:lat` and `geof:lon` that would always return the named coordinate. These functions are implemented in our **JenaX** extension. Such functions are also suggested in the GeoSPARQL issue 160.[18] Additionally, some applications require retrieving the latitude and longitude of a point as separate columns. With **GeoSPARQL 1.1**, the functions `geof:maxX` and `geof:maxY` could be (ab)used to retrieve the coordinates from a point, but knowledge of the coordinate order is required.

4. Performance optimisations in Apache Jena

Some system-specific (targeted) performance improvements were realised on a technical level, which are reported below.

4.1. Per-graph geo-index

In our use case, we have several named graphs, for example, some of which contain disaster events, others which contain administrative geo-boundaries of countries in the world, etc.. For certain use cases, we know exactly the set of graphs over which we want to query. If the question is to know which country an event is located in, it is enough to search only through the administrative regions graph. By default, the Apache Jena geo-index is global across the whole data set. So we implemented an option to create a geo-index by named graphs. For use cases with multiple graphs containing spatial data, this can provide a significant performance boost. The results of our benchmark are displayed in the following table. We tested the approach with an artificial geo-spatial data set consisting of 16 graphs with 262 144 geo-spatial objects in each and a `geof:sfIntersects` spatial query.

	Original impl.	Improved impl.	Matching objects	Speed-up/slow-down
Query a single named graph	35 seconds	5 seconds	1 024	7×
Query all named graphs …				
… using `GRAPH ?g {...}`	5.8 minutes	1.4 minutes	16 384	4×
… using union default graph	1.4 minutes	1.4 minutes		1×

4.2. Improved geo-index serialisation for faster startup time

During our project, we were hampered by the long start-up time of Apache Jena Fuseki reading the geo-spatial index file. To that end, we implemented an improved storage format (serialisation, deserialisation) of the database geo-index on disk. To test the effectiveness of our new implementation, a small and a larger artificial geo-spatial data set was generated containing either 16 named graphs with 512×512 polygon tiles, or 32 named graphs with 1024×1024 polygon tiles each, spread equally across the globe.

		Original impl.	Improved impl.	Speed-up/slow-down
Small geo-index	build time	1.4 minutes	1.8 minutes	0.78×
	load time	24 seconds	2 seconds	12×
Large geo-index	build time	9 minutes	12 minutes	0.75×
	load time	3 minutes	8 seconds	22×

While it takes longer to build the index, the start-up times are much faster afterwards, which was an important improvement for our project.

[17] https://yasgui.triply.cc/

[18] https://github.com/opengeospatial/ogc-geosparql/issues/160

Figure 7: The whole world is America (notice additionally the islands west of Alaska wrapping around the globe)

4.3. Updating the geo-index

Currently, the geo-index in Apache Jena is static and built at start-up. This means it unfortunately does not support SPARQL `UPDATE` or `LOAD` statements to modify the data contained in the database (at least not without restarting). In our project, we had frequent updates to geo-spatial data. While the ideal goal would be to have a self-updating geo-index, for now, we have implemented a method to update the geo-index manually during the run-time of the Apache Jena Fuseki server. The code is available as a Jena Fuseki Module on our GitHub.[19]

4.4. Manually optimising geo-spatial queries

At the time of writing, Apache Jena is not capable of any advanced query optimisation for GeoSPARQL queries. For example, to find all geo-spatial objects contained within the United States, a naive query might look like this:

```
<https://data.coypu.org/country/USA/geometry/boundary> geo:sfContains ?object .
```

On our administrative boundaries data set with 49 535 geo-spatial objects, this query takes about 40 seconds. The reason is that the geographical region of the United States includes overseas regions, so when the server implementation is looking for objects contained within the US, it will first look up all objects in the geo-index using the bounding box of that multi-polygon, and that effectively spans the whole world (see Figure 7).

Ideally, the GeoSPARQL database could detect such situations and do query optimisation for them. For our case, we have chosen the manual way of providing some additional functions to work with the queries: `spatial:st_dump` can be used to break down a multi-polygon into its parts, and `spatial:withinBoxMultipolygonGeom` can be used to make such a geo-index lookup. Our manually tuned query is thus

```
VALUES ?boundary { <https://data.coypu.org/country/USA/geometry/boundary> }
?boundary geo:asWKT ?boundaryGeom .
?feature spatial:withinBoxMultipolygonGeom (?boundaryGeom) .
?feature geo:hasGeometry ?object .
?boundary geo:sfContains ?object .
```

and only takes 8 seconds to execute.

These functions are available in our **JenaX** extension. **GeoSPARQL 1.1** has some basic support for the deconstruction of multi-polygons: `geof:geometryN(?geom, ?geomIndex)` can be used to extract the *N*th geometry and `geof:numGeometries` can be used to query the total number of geometries. However, using these functions is cumbersome at best and will require multiple queries and mechanical construction of queries. Hence we propose to add a property function to destructure geometry collections at GeoSPARQL issue 399.[20]

[19] https://github.com/AKSW/fuseki-mods/tree/adaptions/jena-fmod-geosparql
[20] https://github.com/opengeospatial/ogc-geosparql/issues/399

5. GeoSPARQL 1.1 function summary

The following lists give an overview of the new functions in GeoSPARQL 1.1 and their availability in Apache Jena and the JenaX extension.

Classes New classes specified:
geo:GeometryCollection, geo:FeatureCollection.
(These can be used to model geometry collections using RDF.)

Properties

Generic informative properties
geo:has$^{(M)}$Size, geo:has$^{(M)}$Length, geo:has$^{(M)}$PerimeterLength, geo:has$^{(M)}$Area, geo:has$^{(M)}$Volume.

Properties to connect specific geometries to a feature
geo:hasCentroid, geo:hasBoundingBox.

Properties for resolution and accuracy information
geo:has$^{(M)}$SpatialResolution, geo:has$^{(M)}$SpatialAccuracy.

Properties for geometry representation
geo:asGeoJSON, geo:asKML, geo:asDGGS.

Functions

GeoSPARQL 1.1	J	JX
geof:metricDistance	1	
geof:metricBuffer		
geof:concaveHull		
geof:boundingCircle		
geof:centroid		✓
geof:dimension	✓	
geof:coordinateDimension	✓	
geof:spatialDimension	✓	
geof:geometryType		
geof:is3D		
geof:isEmpty	✓	
geof:isMeasured		
geof:isSimple	✓	
geof:transform	2	
geof:asWKT geof:asGML	3	
geof:asGeoJSON	✓ ↑	
geof:asKML geof:asDGGS		
geof:$^{(m)}$length		✓
geof:$^{(m)}$perimeter		✓
geof:$^{(m)}$area		✓
geof:geometryN		4
geof:numGeometries		4
geof:max{X,Y,Z}		
geof:min{X,Y,Z}		

Aggregate functions

GeoSPARQL 1.1	J	JX
geof:aggBoundingBox		5
geof:aggBoundingCircle		
geof:aggCentroid		5
geof:aggConcaveHull		
geof:aggConvexHull		5
geof:aggUnion		✓

Data types

geo:geoJSONLiteral		✓
geo:kmlLiteral		
geo:dggsLiteral		

Non-standard functions reference:
1 spatialF:distance*
2 spatialF:transformSRS
3 spatialF:transformDatatype
4 spatial:st_dump*
5 geof:collect*

J Availability in Apache Jena
JX Availability in JenaX extension
1–5 Non-standard function
$^{(M)(m)}$ a metric variant of this function is also described, e.g. `geof:`$^{(m)}$`area` → `geof:area` and `geof:metricArea`
↑ this function was upstreamed from JenaX
* these functions are not the same but can be used for a similar purpose

Further functions that might be interesting for the GeoSPARQL community that are not part of the standard but available in Jena or JenaX include:

- functions to create new objects such as `spatialF:convertLatLon`[J], `geof:makeLine`[JX], and `spatialF:convertLatLonBox`[J],
- functions `geof:simplifyDp`[JX] and `geof:simplifyVw`[JX] to simplify polygons,
- `geof:collect`[JX] to create a geometry collection, and `spatial:st_dump`[JX] to destructure a geometry collection into its parts,
- `geof:lat`[JX] and `geof:lon`[JX] to unambiguously address the latitude and longitude of a point,
- a range of functions to work with the H3 DGGS, with the possibility to specify the desired cell resolution when approximating existing geometries (JenaX),
- and some functions to quickly retrieve objects from the geo-index using `spatial:intersectBoxGeom`[J], `spatial:withinBoxGeom`[J], `spatial:withinCircleGeom`[J], and `spatial:withinBoxMultipolygonGeom`[JX].

6. Future topics in GeoSPARQL

Local coordinate reference systems are being discussed in GeoSPARQL issue 500.[21] They are an important requirement for modelling local spatial data that is not rooted in any particular location on the Earth and can operate in Euclidean space rather than on the ellipsoid. Examples of such usage include storage warehouse robots, architecture and building planning, 3D-printing machines, or toy building block instructions. Approaches for temporal RDF modelling have been proposed already in 2007 [18]. The connection between spatial and temporal data should be strengthened beyond the remark in the GeoSPARQL standard that "query extension functions for spatio-temporal operations are not present in the GeoSPARQL standard" [12]. Finally, the modelling of three-dimensional objects using GeoSPARQL seems to be not well understood yet. The Apache Jena GeoSPARQL implementation does not currently support working with 3D objects. There is also GeoSPARQL issue 429[22] which is discussing this topic.

7. Conclusion

As has been shown, there is a significant advantage of using a standardised vocabulary like GeoSPARQL to process spatial data with RDF. Furthermore, GeoSPARQL 1.1 brings many advantages compared to the first version of the standard. We are interested in contributing additional GeoSPARQL 1.1 support to Apache Jena and have already started to do so. Yet, GeoSPARQL 1.1 is still not enough for some use cases. We believe some extensions may be interesting to the larger community and thus it would be beneficial to have them standardised. In that regard, we have already begun to open some issues on the GeoSPARQL issue tracker.

Software and data availability

The source code for this work is available on our GitHub:

- Apache Jena changes as described in Section 4: https://github.com/AKSW/jena/tree/coypu
- Online geo-index module for Apache Jena: https://github.com/AKSW/fuseki-mods/tree/adaptions/jena-fmod-geosparql
- JenaX extension module: https://github.com/Scaseco/jenax
- RdfProcessingToolkit: https://github.com/SmartDataAnalytics/RdfProcessingToolkit [19]

[21] https://github.com/opengeospatial/ogc-geosparql/issues/500
[22] https://github.com/opengeospatial/ogc-geosparql/issues/429

- GridBench generator: https://github.com/AKSW/gridbench
- GridBench results: https://github.com/AKSW/gridbench-results (uses the Linked SPARQL Queries framework [20] for producing benchmark results in RDF)

We used the following Docker images for our experiment:

- Apache Jena with changes: aksw/fuseki-geoplus:5.0.0-1
- Apache Jena without changes: aksw/fuseki-vanilla:5.0.0-1

These RDF data sets were loaded in the experiments:

- geoBoundaries [21] data set as RDF:
  https://raw.coypu.org/static-data/static/geoboundaries/cgaz_adm0.ttl.bz2,
  https://raw.coypu.org/static-data/static/geoboundaries/cgaz_adm1.ttl.bz2,
  https://raw.coypu.org/static-data/static/geoboundaries/cgaz_adm2.ttl.bz2
- MoIn data set: https://maven.aksw.org/archiva/#artifact/org.aksw.moin/moin/1.20220502.0
- CoyPu project OSM River demo: https://maven.aksw.org/archiva/#artifact/org.coypu.thirdpary.osm/osm-river-planet/2022-10-05

Acknowledgments

The authors acknowledge the financial support by the German Federal Ministry for Economic Affairs and Climate Action in the project CoyPu (project number 01MK21007A) and by the German Federal Ministry for Digital and Transport in the Project Moby Dex (project number 19F2266A).

All map screenshots from OpenStreetMap.[23]

References

[1] C. Stadler, J. Lehmann, K. Höffner, S. Auer, LinkedGeoData: A core for a web of spatial open data, Semantic Web 3 (2012) 333–354.

[2] H. Bast, P. Brosi, J. Kalmbach, A. Lehmann, An efficient RDF converter and SPARQL endpoint for the complete OpenStreetMap data, in: Proceedings of the 29th International Conference on Advances in Geographic Information Systems, 2021, pp. 536–539.

[3] M. Hofer, D. Obraczka, A. Saeedi, H. Köpcke, E. Rahm, Construction of knowledge graphs: State and challenges, 2023. arXiv:2302.11509.

[4] B. Bucher, E. Folmer, R. Brennan, W. Beek, E. Hbeich, F. Würriehausen, L. Rowland, R. A. Maturana, E. Alvarado, R. Buyle, et al., Spatial linked data in Europe: Report from spatial linked data session at Knowledge Graphs in Action, Official Publication-EuroSDR (2021).

[5] E. Folmer, Linking authoritative (government) data with community data (e.g. Wikidata), in: EuroGeographics and EuroSDR, 2023, pp. 24–25.

[6] A. Becker, A. Ahmed, M. A. Sherif, A.-C. Ngonga Ngomo, Cobalt: A content-based similarity approach for link discovery over geospatial knowledge graphs, in: Knowledge Graphs: Semantics, Machine Learning, and Languages, IOS Press, 2023, pp. 177–193.

[7] C. Zinke-Wehlmann, A. Kirschenbaum, Geo-L: Topological link discovery for geospatial linked data made easy, ISPRS International Journal of Geo-Information 10 (2021).

[8] B. Yaman, K. Thompson, R. Brennan, Standards conformance metrics for geospatial linked data, in: Knowledge Graphs and Semantic Web: Second Iberoamerican Conference and First Indo-American Conference, KGSWC 2020, Mérida, Mexico, November 26–27, 2020, Proceedings 2, Springer, 2020, pp. 113–129.

[23] https://www.openstreetmap.org/copyright

[9] S.-A. Kefalidis, D. Punjani, E. Tsalapati, K. Plas, M. Pollali, M. Mitsios, M. Tsokanaridou, M. Koubarakis, P. Maret, Benchmarking geospatial question answering engines using the dataset GeoQuestions1089, in: International Semantic Web Conference, Springer, 2023, pp. 266–284.

[10] T. Osman, G. Albiston, GeoSPARQL-Jena: Implementation and benchmarking of a GeoSPARQL graphstore, in: 23rd European Conference on Knowledge Management Vol 2, 2022.

[11] M. Jovanovik, T. Homburg, M. Spasić, A GeoSPARQL compliance benchmark, ISPRS International Journal of Geo-Information 10 (2021) 487.

[12] N. J. Car, T. Homburg, M. Perry, F. Knibbe, S. J. Cox, J. Abhayaratna, M. Bonduel, P. J. Cripps, K. Janowicz (Eds.), OGC GeoSPARQL–A Geographic Query Language for RDF Data, Version 1.1, 22-047r1, 2024. URL: http://www.opengis.net/doc/IS/geosparql/1.1.

[13] J. R. Herring (Ed.), OpenGIS Implementation Standard for Geographic information – Simple feature access – Part 1: Common architecture, OGC 06-103r4, 2011. URL: https://www.ogc.org/standard/sfa/.

[14] C. Stadler, S. Bin, L. Bühmann, N. Radtke, K. Junghanns, S. Gründer-Fahrer, M. Martin, Semantification of geospatial information for enriched knowledge representation in context of crisis informatics, in: Proceedings of the International Workshop on Data-driven Resilience Research 2022, volume 3376 of *CEUR Workshop Proceedings*, Leipzig, Germany, 2022.

[15] N. J. Car, T. Homburg, GeoSPARQL 1.1: Motivations, details and applications of the decadal update to the most important geospatial LOD standard, ISPRS International Journal of Geo-Information 11 (2022) 117.

[16] B. Bondaruk, S. A. Roberts, C. Robertson, Assessing the state of the art in Discrete Global Grid Systems: OGC criteria and present functionality, Geomatica 74 (2020) 9–30.

[17] D. Habgood, T. Homburg, N. J. Car, M. Jovanovik, Implementation and compliance benchmarking of a DGGS-enabled, GeoSPARQL-aware triplestore, in: Proceedings of the 5th International Workshop on Geospatial Linked Data co-located with ESWC, volume 3157 of *CEUR Workshop Proceedings*, 2022.

[18] M. Koubarakis, K. Kyzirakos, Modeling and querying metadata in the semantic sensor web: The model stRDF and the query language stSPARQL, in: Extended Semantic Web Conference, Springer, 2010, pp. 425–439.

[19] C. Stadler, L. Bühmann, L.-P. Meyer, M. Martin, Scaling RML and SPARQL-based knowledge graph construction with Apache Spark, in: 4th International Workshop on Knowledge Graph Construction @ ESWC 2023, volume 3471 of *CEUR workshop proceedings*, Hersonissos, Greece, 2023.

[20] C. Stadler, M. Saleem, A.-C. Ngonga Ngomo, LSQ framework: The LSQ framework for SPARQL query log processing., in: QuWeDa@ ISWC, 2022, pp. 49–64.

[21] D. Runfola, A. Anderson, H. Baier, M. Crittenden, E. Dowker, S. Fuhrig, S. Goodman, G. Grimsley, R. Layko, G. Melville, et al., geoBoundaries: A global database of political administrative boundaries, PloS one 15 (2020) e0231866.