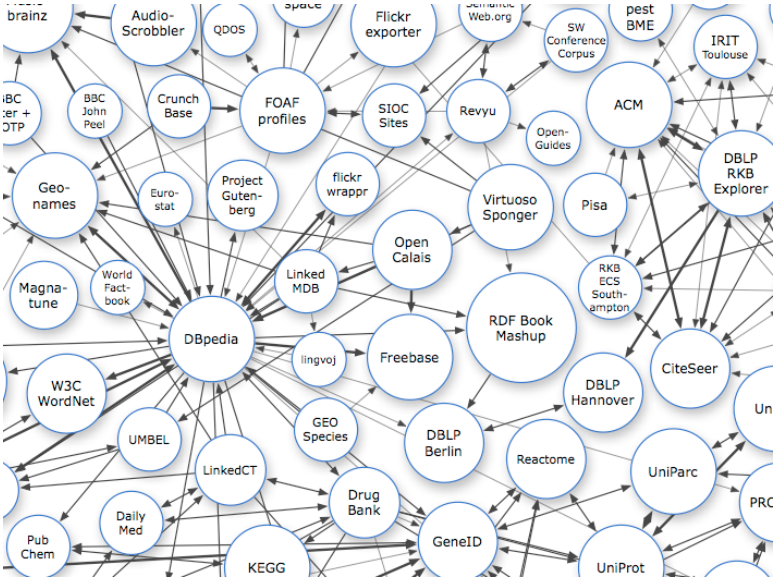
University of Zurich[UZH]

Thesis December 13, 2014

Linked Raster Data



Schüpfer Florian
of Willisau LU, Switzerland

Student-ID: 10-718-328
flo.schuepfer@bluewin.ch

Advisor: **Dr. Thomas Scharrenbach**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

I would like to thank all those who supported me during my study time and my bachelor thesis.
A special thank goes to my advisor Thomas Scharrenbach, who always supported me with his professional and technical knowledge during my thesis. He often triggered new thoughts with his ideas during interesting discussion sessions.
I also want to thank my parents, who enabled me to study and who have always supported me in my life.

Zusammenfassung

Das Semantic Web und Linked Data eröffnen uns riesige Möglichkeiten um Wissen aus verschiedenen Domänen zu integrieren. In der räumlichen Domäne existieren bereits Projekte wie linkedgeodata.org oder GeoSPARQL, welche räumliche Daten mit georeferenzierten Ressourcen verbinden. Diese Projekte operieren alle auf Vektordaten.

In dieser explorativen Arbeit diskutieren wir die Unterschiede zwischen der Integration von Vektor- und Rasterdaten im Semantic Web. Weiter werden wir Methoden zur Verlinkung von Rasterdaten mit georeferenzierten Ressourcen in der SPARQL Abfragesprache finden, diskutieren und Implementieren. Wir zeigen wie geographische Operationen auf Rasterdaten in RDF beschrieben werden können und wie Rasterdaten mit Hilfe des WMS Protokols von Remoteservern geladen werden können.

Wir werten unseren Ansatz aus, in dem wir die Ausführungszeit von verschiedenen Abfragen in verschiedenen Konfigurationen messen. Wir finden heraus, dass die Abfrage auf dem Remote Endpoint am meisten Zeit in Anspruch nimmt und dass wir so wenig Resultate wie möglich vom Remote Endpoint holen sollten, um die Ausführungszeit zu reduzieren.

Am Schluss dieser Arbeit stellen wir fest, dass wir die definierten Ziele erfüllen konnten, obwohl wir wegen der SPARQL Engine, die wir benutzt haben, ein paar Workarounds finden mussten.

Abstract

The Semantic Web and Linked Data open huge possibilities for the integration of knowledge from different domains. In the spatial domain, there are already approaches like linkedgeodata.org and GeoSPARQL, which integrate spatial data with georeferenced entities. These projects operate on vector data like polygons.

In this explorative work, we discuss the differences between the integration of vector and raster data into the Semantic Web. Further we find, discuss and implement a method for linking raster data to georeferenced entities in the SPARQL query language. We show how geographic operations on raster data can be described in RDF and how we can load raster files from remote servers by implementing service calls using the WMS protocol.

We evaluate our approach by measuring and comparing the execution time of different queries in different configurations and find that the largest bottleneck of Linked Raster Data queries is the remote endpoint and that we should fetch as few results from the remote endpoint as possible to reduce the query execution time.

At the end of this thesis, we conclude that we achieved our goals defined at the beginning, altough we had to find some workarounds because of the SPARQL engine we used.

Table of Contents

1

Introduction

When you click on a link in an html document in the World Wide Web of today, you get to another document. When you enter a search term into a search engine in the internet, you will get a list of links of documents that contain the search term you entered before. But when we have a link between two documents, the meaning or semantic of this link is not defined. The relation in which the two documents stand is not explicit. Consider, for example, a page about vehicles. This page will contain a lot of links to different classes of vehicles like cars or motorcycles. Of course, for a human agent, it is clear that a car is a subclass of vehicles. But the search engine has no knowledge of this relation and thus, a search engine is not capable of explicitly answering a simple question like "Does a car belong to the class of vehicles?" or "Is a car a vehicle?".

In the last decade there was a lot of research concerning the Semantic Web and Linked Data. The idea of Linked Data is exactly to overcome the shortcomings we have mentioned above. The goal of Linked Data is to reflect the relations (or meaning) of two resources in the structure of the data itself.

When we consider the example from above, we could state that every car is a subclass of the class vehicle. Different Knowledge Representations (KR) have been put on top of Linked Data such as RDFS and OWL2. Using such a KR we can express the statement from above as:

```
ex:Car rdfs:subClassOf ex:Vehicle
```

Listing 1.1: A Linked Data statement

The (abstract) data model of the Semantic Web is the Resource Description Framework (RDF). In RDF all knowledge is represented in triples of subject, predicate and object [1]. The subject is a resource like the car in the example. The predicate, in this case `rdfs:subClassOf`, describes the relation between the car and the vehicle which is in this case the object of the statement. Each such statement represents a piece of knowledge that can be evaluated by a SPARQL engine.

1.1 Linked Data and geographic entities

Linked Data is especially useful for linking resources together that belong to different domains. An example is the linking of geographic entities to vector data like polygons.

LinkedGeoData "LinkedGeoData is an effort to add a spatial dimension to the Web of Data / Semantic Web" [2]. It uses the database of openstreetmap.org [3] to create a large spatial knowledge base and link this knowledge to resources of other domains like cities or countries. An example could be to reference a polygon representing the boundary of a city like zurich to the resource `dbpedia.org/resource/zurich` which contains other information about the city like the population or neighbour towns.

GeoSPARQL GeoSPARQL extends the SPARL protocol by allowing the user the integration of geographic information about resources in his query. The user,for example, can retrieve resources that are located nearby a specific georeferenced point like in the query from Listing 1.2.

```
PREFIX spatial:<http://jena.apache.org/spatial#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX geo:<http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX gn:<http://www.geonames.org/ontology#>


Select * WHERE {
?object spatial:nearby(40.74 -73.989 1 'mi').
?object rdfs:label ?label
}
LIMIT 10
```

Listing 1.2: GeoSPARQL example query

The expression "spatial:nearby(40.74 -73.989 1 'mi')" is in this case not just a predicate of the objects we search. It is a function call that is executed on the server and searches for resources in the database that are nearby the point (40.74 -73.989) in a distance of 1 mile. This function call is expressed as a Basic Graph Pattern in the query. When the expression "spatial:nearby" is evaluated from the SPARQL engine, the function, to which "spatial:nearby" points to, is executed in the background. We will also use this approach to specify our raster operations.

Maybe we want to find resources that lay in certain geographic region bounded by a polygon. Than we can write the query from Listing 1.3.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/geosparql/function/>


SELECT ?what
WHERE {
  ?what geo:hasGeometry ?geometry .
  ?geometry geo:asWKT ?wkt .


  FILTER(geof:within(?wkt,
     "POLYGON((
-77.089005 38.913574,
-77.029953 38.913574,
-77.029953 38.886321,

-77.089005 38.886321,
-77.089005 38.913574
))"^^geo:wktLiteral))
}
```

Listing 1.3: GeoSPARQL example query

We first state that the object we search for has a geometry. Then we transform the geometry into a literal in the second statement. Like before, the `geof:within` expression represents a function call, that filters our resultset of the first two statements and returns only resources for which the geometry (for example the bounding polygon of a city) lays inside the provided boundingbox.

1.2 Goals and Outline

In the last paragraph, we have seen, how we can link vector data like polygons to geographic entities like cities. In this thesis, we look at an approach for linking raster data to geographic entities.

We discuss two approaches for linking raster data to georeferenced entities in a similar manner: explicit features and feature descriptions. In this explorative work, we implement the feature description approach. We find, discuss and implement methods for describing and executing georeferenced functions on raster data by extending the SPARQL query language. Further we show how to link raster data to georeferenced entities on remote endpoints and how we can load raster files from remote servers by implementing service calls using the WMS protocol.

The main goals defined for this thesis are:

- To assemble a dataset of raster data, preferably of Swiss municipal, cantonal and federal resources.
- To connect the data sets with existing Linked Open Data (LOD), such as GeoNames and dbPedia in order to link entities from the Linked Open Data into the

raster data.

- To assemble statistical data, preferably from Swiss municipal, cantonal and federal offices, and link these to Linked Open Data and the raster data.

First, we will look at the fundamentals of Linked Raster Data and geooperations in greater detail in Chapter one.

We will outline our approach of implementing Linked Raster Data in Chapter three. For the implementation, we use the rdflib framework as our SPARQL engine. With help of the lrdgdal framework, we expose our raster files as RDF graphs, such that we can query the metadata of each raster and its bands. We also describe the semantics and parameters of our extension functions in RDF.

For loading rasters from remote endpoints, we decide to use the WMS protocol. We use the GetMap request to specify the service call. By providing an additional polygon, we can restrict the returned raster file to the desired area. In Chapter five, we also discuss how we could retrieve raster files by category.

In Chapter four, we run some basic usecases of Linked Raster Data queries in different configurations and compare the results. In Chapter five, we will look back at the goals we stated before and discuss our findings. We also discuss the limitations of our application and what still needs to be done in the future.

2

Methods in general

2.1 Linked Data

Linked Data is a recommendation how to publish links to and between data entities, called resources. In the RDF data model, links are established by triples (s, p, o) where p is a qualified binary predicate and o and p are resources. Each of these are uniquely identified by its IRI or anonymous identifier. The instance data and the schemata are both expressed in RDF [4].

In SPARQL, queries are represented as graph patterns that are matched against one or more RDF graphs. The results are obtained by evaluation of these patterns. Further constraints on the results are put by FILTER and JOIN operations. The results are presented by lists of n-tuples or by graphs [4].

In SPARQL, the query predicate itself can be a variable. The evaluation of the Basic Graph Pattern `map:poly1 ?prop map1:poly2`, for example, contains all links that exist between two particular polygons in a map. Evaluating the query variable `?prop` will provide us with all those properties that exist between the two polygons, for example, topological relations expressed by `rcc:touches` or `rcc:tangentialProperPartOf`.

RDF triples can be universal or valid within a certain context only. In the first case, they are part of the default graph whereas they are part of a named graph in the latter case. Named graphs are referred to by an IRI and can be queried for explicitly by using SPARQL [4].

2.2 Linked Geo Data

Linked Data can be helpful in finding geographic entities. Consider, for example, you want to find all nature reserves in Switzerland with at least 20 percent forest area and all intersecting municipalities. Linked Data provides metadata for layers and geographic entities of a GIS. These metadata can then be linked to other Linked Data of other domains (graphs) and is thus building a bridge between different knowledge domains [4].

In the example above, we would treat each object `o` in the layer containing nature reserves as an RDF resource and identify them with the type of a nature reserve. This is expressed by the pattern: `?o rdf:type dbpedia:naturereserve`. For the results

of the query evaluation, there exist other links to other resources in other domains. For example, the different areas lay in different municipal districts of the canton which themselfs have a lot of links. To get only those areas with at leat 20 percent forest, we have to calculate the forest area. Since the forest area is most probably a categorical raster file, we need Linked Raster Data [4].

2.3 Linked Raster Data

Raster Data represent data fields. These are either categorical fields, such as the forest coverage or continous fields, such as elevation or slope.

Vector data describes geometric objects, such as polygons and lines. Vector data can be easyly represented in RDF by assigning every object an IRI. For raster data, it is not that simple. In a raster, objects only exist in a certain context. The entities of a raster are just pixel values that represent certain features. Examples would be categorical features such as rivers or roads or continous features such as elevation or temperature [4].

When we want to implement Linked Raster Data, we can either use explicit features (e.g. rasterized geo-objects) or feature descriptions [4].

Explicit features can be directly represented in RDF or GML. We only need a proper way of feature extraction. For example, a mountain can be represented as a set of pixels with an IRI applied to it, such that it can be linked to other resources. In this approach, there exists the problem of proper bounding of the object in the raster. For example, it isn't clear where the mountain begins and ends in an elevation raster. Another approach could be to identifiy an object by the position of the corresponding cells in the raster and store only the metadata of the raster and the position of the object in RDF [5].

When using feature description to describe objects, we actually describe parametrized geographic operations that calculate the result depending on additional context information or subqueries. In the example before, we have a feature description, because we only want nature reserves with a forest area of at least 20 percent. Consider extending the example with the additional constraint that the resulting areas on average have to have a slope value that is larger than the average slope of all forest areas. In this case, the result can't be computed directly because of the context. Pre-computation is also not possible, because the constraint (above average slope) can change over time.

2.4 Geographic operations

In order to perform Linked Raster Data operations, the raster file has to store metadata about the geographic region it covers and the coordinate system it uses. One important information is the projection of the image. A map projection is necessary to transform the latitude/longitude of a point on a sphere surface into locations onto a plane. There exist a lot of different projections depending on the purpose of the map. Below are two examples: The Figure 2.1 shows a cylindrical projection and Figure 2.2 a azimuthal

(projection on a plane) projection. In the first case, one sees that the spatial distortion gets larger to the poles. In the second case, the map shows distances and directions accurately from the center point, but distorts shapes and sizes elsewhere [6].

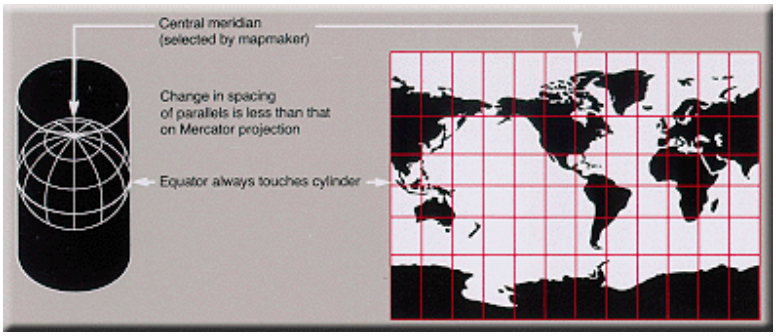


Figure 2.1: A cylidrical mercator projection [7].

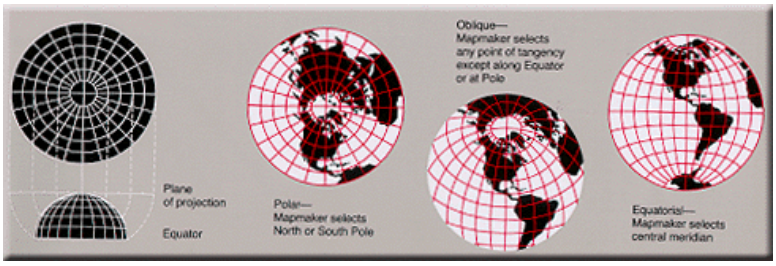


Figure 2.2: An equidistant azymutal projection [8].

The creation of a map projection involves two steps. In the first step a model for the earth is chosen. In most cases this is a sphere or an ellipsoid. Since the earth is not a perfect sphere, some information is always lost. In the second step, the geographic coordinates (longitude/latitude) have to be transformed into cartesian or polar plane coordinates [6].

Every raster file with geographic metadata has therefore a Spatial Reference System with a specific coordinate system (a specific map projection) and supports transformations between different spatial reference systems. Spatial Reference Systems are defined by the OGC (Open Geospatial Consortium) and are expressed in Well Known Text (WKT) literals [9].

One of the most widely used standards is the WGS84 (World Geodetic System) reference system. All coordinates provided and obtained by SPARQL queries are expressed in WGS84 literals by default; other reference systems can, of course, be specified. Since the raster itself can have a different Spatial Reference System, the coordinates of the query have to be transformed into coordinates suitable for the dataset of the raster.

When we want to make operations on the raster by providing geo coordinates, we not only have to convert the geo coordinates into the correct Spatial Reference System, we also have to convert them into raster coordinates. In order to do this, the raster

provides an affine geotransfom. The geotransform is a matrix providing all necessary parameters to make the calculations. For example, in GDAL (see Section 3.1.1), in the case of north-up images (when theres no rotation involved), the geotransform consists of 4 parameters: the pixel width and height (in geo coordinates) and the geo coordinates of the top-left corner [10].

2.5 Feature description

For the feature description approach, we need RDF for describing the context and the parameters of our operations in terms of Basic Graph Patterns.

To achieve the execution of a certain operation in SPARQL, we need to express our function call as a set of Basic Graph Patterns. This means, that we have to identifiy each operation on raster objects as a query predicate with its own IRI. For example, the Basic Graph Pattern `?points gis:lessThan 500` would execute the function to which `gis:lessThan` points to and would in this case return all points on the selected raster that have a lower value than 500.

3

Methods implementation

3.1 Querying a raster file

In order to evaluate SPARQL queries involving raster files, we have to find efficient methods for reading the metadata of the raster file and exposing it as an RDF Graph, such that it can be queried in SPARQL. We also need a method for hooking into the SPARQL engine in order to evaluate our own Basic Graph Patterns. In the following sections, we will outline all the relevant techniques and tools used to achieve these goals.

3.1.1 Reading the metadata of a raster

To open and read a georeferenced raster file, we need the gdal python-bindings from osgeo [11]. GDAL is an opensource translator library for raster and vector geospatial data formats [12]. GDAL provides a convenient interface to obtain general metadata, such as width, height, the number of bands etc, as well as geographic metadata, such as the central longitude/latitude, the vertical and horizontal resolution, the Spatial Reference System and the geotransform (see also Section 2.4) . With the help of gdal, we can represent a raster file as a memory object by mapping the relevant metadata into a python class. This is already implemented in the lrdgdal framework [13].

3.1.2 Exposing a raster as an RDF graph

In the next step we convert the metadata of a raster file into a set of RDF triples. For every raster file, there is one graph (the default graph) that stores the metadata of the raster and one named graph that stores the metadata of a band in the raster. This has also already been implemented in lrdgdal. We illustrate a query that retrievs all bands of a raster with the IRI "http://example.com/raster" in Listing 3.1:

```
PREFIX java: <http://evolizer.org/ontologies/seon/2009/06/java.owl#>
PREFIX gdal: <http://scharrenbach.net/gdal#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
SELECT ?result WHERE {
  <http://example.com/raster> gdal:Band ?b.
}
```

Listing 3.1: An lrdgdal example query

3.1.3 Using the SPARQL engine of rdflib to evaluate Basic Graph Patterns and custom functions

To process SPARQL queries we use rdflib. Rdflib is a Python library for working with RDF. It also comes with a built-in SPARQL-Engine [14].

When we want to evaluate our own custom functions, we can do this by writing our own evaluation function for the Basic Graph Pattern (BGP) matching. From there, we have access to all triples in a basic SPARQL query. We can then define our own predicates in our own namespace and process all triples that match our predicates. With help of these predicates, we can than specify our function calls in triple patterns. All other triples that are not in our namespace are delegated to the query processor of rdflib.

An exampe query, where we retrieve a single value belonging to a specific geocoordinate in the raster for the first band is illustrated in Listing 3.2.

```
PREFIX gdal: <http://scharrenbach.net/gdal#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {

  ?g a gdal:Band .

  Graph ?g {
    ?band_value gdal:Function gdal:GetPixelValue .
    ?band_value gdal:Param1 POINT(47.36, 7.85).
  }
}
```

Listing 3.2: query for retrieving a single pixelvalue

First we define that `?g` is a `gdal:Band` and since all bands are treated as named graphs, we have to execute the query for retrieving the pixel value in the GRAPH-section of the query. We first specify the function we want to execute and then the parameters of the function, which in this case is the georeferenced point. From our custom evaluation function, we can then access the pixeldata of the band from the query context, which is in this case the band graph.

As mentioned before, we have to define our own predicates and objects that are in this case: `gdal:Function`, `gdal:GetPixelValue` and `gdal:Param1`. We will use this

structure to specify all our function calls.

Problems with rdflib and workaround The case with have described above is a SPARQL approach to specify function calls in terms of triple patterns that should reflect the underlying graph structure. This is the ideal case.

In practice, we have experienced several problems with rdflib that prevented us from implementing our queries in the way we described before:

- The query processor of rdflib always evaluates the triples inside the GRAPH-section before all other triples. This prevents us from restricting the graph variable (`?g` in the case before) and thus we are not able to reflect the graph structure of lrdgdal in our queries.
- Another problem is that we can not mix BGPs with other expression like BIND or FILTER. This is also due to the execution order of rdflib. The Basic Graph Patterns are always executed before the other expressions. This is a problem since we may want to use a BIND-expression to execute a custom function and process the result further or we want to use a FILTER-expression to restrict the result set before further processing.

To prevent the problems from above, we would have to seriously modify the framework, which is beyond the scope of this work. Therefore, we had to find a workaround to simulate the execution of our Basic Graph Pattern evaluation.

Custom functions Direct function calls are already implemented on most SPARQL endpoints. One example is the "CONCAT" function for concatenating two strings together [1]:

```
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
SELECT ( CONCAT(?G, "␣", ?S) AS ?name )
WHERE { ?P foaf:givenName ?G ; foaf:surname ?S }
```

Listing 3.3: Concatenation of 2 strings in SPARQL

When we want to store the result of the function as a new variable for further processing, we can do this with the BIND expression [1]:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
    ?P foaf:givenName ?G ;
       foaf:surname ?S
   BIND(CONCAT(?G, "␣", ?S) AS ?name)
}
```

Listing 3.4: Concatenation of 2 strings with BIND expression

Every SPARQL endpoint has a set of built-in functions. At the same time, SPARQL can be extended by providing custom functions. When we want to execute our own functions, we have to extend our SPARQL engine, since by default, rdflib doesnt allow that.

In our custom functions, we can access the query context, that is the dataset, that holds all our rasters and bands. From there, we can access all the data we need for the evaluation.

Consider the example from Listing 3.2, where we queried a pixel value for all bands. With custom functions, we can write the same query like this:

```
PREFIX gdal: <http://scharrenbach.net/gdal#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
  ?r a gdal:Raster.
  BIND(gdal:GetRasterBand(?r, 1) AS ?b1)
  BIND(gdal:GetRasterBand(?r, 2) AS ?b2)
  BIND(gdal:GetRasterBand(?r, 3) AS ?b3)
  BIND(gdal:GetPixelValue(?b1, POINT(47.36, 7.85)) AS ?pv1)
  BIND(gdal:GetPixelValue(?b2, POINT(47.36, 7.85)) AS ?pv2)
  BIND(gdal:GetPixelValue(?b3, POINT(47.36, 7.85)) AS ?pv3)
}
```

Listing 3.5: query for retrieving a single pixelvalue with extension functions

This query does exactly the same thing like the query 3.2. Actually, its not the idea of SPARQL to make excessive use of BIND expressions and custom functions, because the graph structure of our raster can not be reflected in the queries with this approach. However, because of the problems we experienced with rdflib, we had to find a suitable workaround.

In the coming sections and chapters, we will always write our queries in BGP-form but in our implementation, we will simulate the queries by using custom functions.

3.2 Clipping of a raster

When we do calculations on a raster file, we execute operations on a restricted area of the raster. Do define this area we provide a polygon as WKT Literal and generate a binary mask out of it. The masking process is illustrated in Figure 3.1.

With the help of the mask and the bounding box, we generate a new dataset with bands that are restricted to the area in the boundingbox of the mask. The lrdgdal framework already provides methods for the clipping of a dataset.

In order to implement clipping in our application, we can define a new function "ClipRasterToPolygon" which expects a valid IRI of the raster we want to clip, a polygon that defines the clipping region and a name for the new raster. Below is an example query that clips a single (already loaded) raster with the IRI "http://example.com/raster" to a polygon and returns the IRI of the new raster as result.

Figure 3.1: Illustration of the masking process [15].

```
PREFIX gdal: <http://scharrenbach.net/gdal#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {

?iri gdal:Function gdal:ClipRasterToPolygon.
?iri gdal:Param1 "http://example.com/raster".
?iri gdal:Param2 "POLYGON((47.13 7.58), (47.36, 7.54), (47.13 7.58))".
?iri gdal:Param3 "wald".

}
```

Listing 3.6: query for clipping a raster to a polygon

3.3 Intersecting multiple rasters and bands

When we want to execute useful geo operations on raster data, we often have to query multiple maps to obtain the result. Consider the example from Section 2.2, where we wanted to find all nature reserves in Switzerland with at least 20 percent forest area and all intersecting municipalities.

To answer the query, we have to go through the following steps:

1. We first have to take a map that contains all forest areas in Switzerland. We then generate a binary map, that marks every forest pixel on the map with a one and all other pixels with a zero.

2. We intersect the binary map with a map containing all nature reserves of switzerland to obtain the intersecting areas.

3. In the last step, we take the result and intersect it with a map containing all municipalities of Switzerland to obtain the final result.

In the example, we can use different raster files for each map. However, intersecting raster files is not that easy as it may look at the first glance. When we intersect different raster files, we have to take different problems into account:

- The different rasters may not cover the same geographic area. That means, we need to calculate the intersection of the rasters and clip both to the resulting bounding box.
- The different rasters may have different Spatial Reference Systems. This implies that we have to convert them into the same system.
- Raster files can have different spatial resolutions. Therefore, the same geographic area can cover more pixels on one raster, than it covers on another raster.

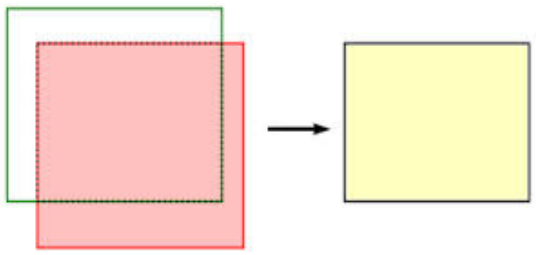


Figure 3.2: Intersection of two rasters [16]

The first two problems are solved, since we already have functions provided in the framework for this tasks. For solving the third problem, we need to interpolate pixel values and convert all rasters to the same resolution. We have multiple ways to do this. Here we will outline two of them:

- We use the gdal_translate command from the gdal framework to convert the whole raster (and thus all bands) into another spatial resolution [12]. Lets assume that we have a raster from a file called "forest.tif" and we want to convert it to the resolution 800 X 600. Then we would execute the following command from the gdal-bin library:

```
gdal_translate -of GTiff -outsize 800 600 forest.tif forest2.tif
```

Listing 3.7: A gdal_translate command

  This will create a new raster file with a different spatial resolution that can be loaded into our dataset as a new graph.

- Another approach is to only convert the raster arrays themselves if we have binary operations involving another rasters with other resolutions. We can do this with the "zoom" function of scipy when we use numpy arrays to store our raster data [17].

A downsampling example with two arrays A and B with different resolutions looks like the following:

```
scaleX = float(B.shape[0]/A.shape[0])
scaleY = float(B.shape[1]/A.shape[1])
out = scipy.ndimage.interpolation.zoom(A, [scaleX, scaleY])
```

Listing 3.8: A downsampling example

When using the first approach, we always have to convert the whole raster, even if we only need a single band. The creation of a new file each time we have to rescale the raster is also a large overhead. When we use the second method, we dont have these problems and we can convert the arrays when we need to. Therefore, we decided in favour of the second approach. However, when we want to convert the resulting array of a map operation back to GeoTiff, we have to take into account that the spatial resolution changed and we have to recalculate the geotransform manually.

We could ommit all problems resulting from the intersection of different rasters when we only work with bands inside the same raster file. This way, we would have no problems with different resolutions, spatial reference systems and geographic areas. However, this approach proved as not suitable for our task, since we want to be able to load different raster files from different servers (see also Section 3.7) and intersect them to perform our calculations. With one raster, this is not possible, because we would have to "pack" all maps we need manually into the same raster file, which is not a dynamic and practical approach for the user of our endpoint. Therefore we decided against this approach.

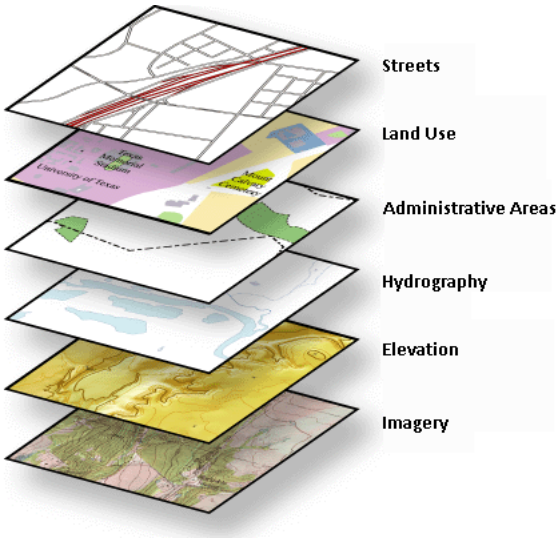


Figure 3.3: Illustration of multiple bands of a raster [18].

3.4 The structure of Linked Raster Data queries

When we construct SPARQL queries involving raster data, we go through the same steps:

1. We obtain our georeferenced points or possibly polygons for which we want to calculate something on the raster either locally or from a remote endpoint. This step is optional.
2. We select the rasters and bands on which we want to calculate.
3. We get the data (the pixel values) of the selected bands.
4. We perform the actual operations, such as projections and aggregations, on the matrices of the bands.
5. In the last step, we return the result in a useful format for the user (optional).

In the last step, we have to convert the resulting matrix (the pixel values) back into a raster file format like GeoTiff.
In order to be able to convert the matrix into a raster file, we have to get the relevant metadata of the raster (like SRS, GeoTransform, width and height). We can obtain these values when we reference the raster belonging to the pixelvalues we have obtained. From this raster we can then take the Spatial Reference System. The geotransform has to be recalculated since the resolution of the arrays could have changed. A simple query that adds two bands of a single raster file (that is already loaded) with the IRI "http://example.com/raster" and returns the result as Base64-encoded GeoTiff file, is illustrated in Listing 3.9.

```
PREFIX java: <http://evolizer.org/ontologies/seon/2009/06/java.owl#>
PREFIX gdal: <http://scharrenbach.net/gdal#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
SELECT ?result WHERE {

  ?b1 a gdal:Band
      gdal:BandNumber 1.

  ?b2 a gdal:Band.
      gdal:BandNumber 2.

  GRAPH ?b1{
     ?pv1 gdal:Function gdal:GetPixelValues.
  }
  GRAPH ?b2{
     ?pv2 gdal:Function gdal:GetPixelValues.
  }

  ?add gdal:Function gdal:ADD.
       gdal:Param1 ?pv1.
       gdal:Param2 ?pv2.

  ?result gdal:Function gdal:CreateGeoTiff.
          gdal:Param1 "http://example.com/raster".
          gdal:Param2 ?add.
}
```

Listing 3.9: Addition of 2 bands

As mentioned, we select the bands on which we want to perform our calculations. Therefore we first provide the IRI of the raster from which we want to access the bands. The second parameter is the index of the raster.

In the next step, we want to select the pixelvalues of the bands.

In the last section of the query, we finally perform our calculations on the pixelvalues and return our result as a GeoTiff - string. For the CreateGeoTiff - function we also provide the iri of the raster from which we want to copy the metadata.

3.5 Describing geo operations in RDF

In the last section, we discussed the structure of Linked Raster Data queries. We saw, that we have to implement different functions in order to process useful queries.

In a real world application, we want to send our query to a SPARQL endpoint anywhere in the World Wide Web. Since our implementation of Linked Raster Data is only available on our own Endpoint, we have to provide a mechanism that allows the user to lookup all available extension functions and their descriptions on our server. We have to provide all information necessary for the user to be able to construct his query.

We could provide all information necessary to use our endpoint on a website or in a textfile, but that wouldn't be a very dynamic or userfriendly approach. Consider for example, that we have multiple endpoints that provide Linked Raster Data functions and the user wants to find the most suitable for his needs. It would be very cumbersome to search multiple websites or files to get all information necessary. It would be much easier to just send a SPARQL query to different endpoints and compare the results. Therefore, we have to describe our functions in RDF and add the resulting triples to our Default Graph of the raster.

Meta Data In order to be able to describe our functions, we need a vocabulary for all relevant properties that functions have.

When we want to describe subjects in a given domain, we use the same vocabulary to describe them in RDF. For example, when we want to describe the country in which some resource is located, we use the predicate `dbo:country`.

Since there doesn't exist an implementation of Linked Raster Data that operates on feature descriptions yet, we will just add predicates and objects in our own namespace for our function descriptions.

The most important properties of a function are:

- The parameters of the function
- The name of the function
- A description of what the function does
- All parameters of the function and its result
- The datatypes of the parameters and the result

The description of our vocabulary in RDF is shown in Listing 3.10.

```
gdal:PixelArray rdf:type gdal:Parameter.
gdal:Number rdf:type gdal:Parameter.
gdal:Integer rdf:type gdal:Parameter.
gdal:Base64 rdf:type gdal:Parameter.
geo:long rdf:type gdal:Parameter.
geo:lat rdf:type gdal:Parameter.
gdal:Polygon rdf:type gdal:Parameter.
gdal:Radius rdf:type gdal:Parameter.
gdal:Raster_IRI rdf:type gdal:Parameter.
gdal:Band_IRI rdf:type gdal:Parameter.

gdal:PixelArray gdal:Datatype gdal:NumpyArray.
gdal:Number gdal:Datatype xsd:float.
gdal:Integer gdal:Datatype xsd:Integer.
gdal:RasterBand gdal:Datatype gdal:Band.
gdal:Base64 gdal:Datatype xsd:base64Binary.
geo:long gdal:Datatype geo:wktLiteral.
geo:lat gdal:Datatype geo:wktLiteral.
gdal:Radius gdal:Datatype xsd:float.
gdal:Polygon gdal:Datatype geo:wktLiteral.
gdal:Raster_IRI gdal:Datatype xsd:string.
gdal:Band_IRI gdal:Datatype xsd:string.
```

Listing 3.10: Description of the meta data

First we introduce all parameters we use in our functions and assign them the type of `gdal:Parameter`. Then we describe the datatype of all parameters. For types, that are already described in XML or GeoSparql, we use the appropriate namespace. For all other types, we just introduce a datatype in the `gdal`-namespace.

The next step is the description of the actual functions. A block for one function is shown in Listing 3.11.

```
gdal:less_equals rdf:type gdal:Function.
gdal:less_equals gdal:Description "Returns a binary matrix with 'True' for
    all values smaller or equal than the provided threshold and 'False'
    otherwise".
gdal:less_equals gdal:FunctionName "less_equals".
gdal:less_equals gdal:Param1 gdal:PixelArray.
gdal:less_equals gdal:Param2  gdal:Number.
gdal:less_equals gdal:Output gdal:PixelArray.
```

Listing 3.11: description of the "less_equals" function

As mentioned before, we have to introduce predicates to describe the most important properties of our functions. For the description of the parameters and the output, we use the names from above.

When we want to find out all relevant information of the function with the name "less", we would write the query illustrated in Listing 3.12

```
PREFIX java: <http://evolizer.org/ontologies/seon/2009/06/java.owl#>
PREFIX gdal: <http://scharrenbach.net/gdal#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
SELECT * WHERE {
  ?f a gdal:Function;
     gdal:FunctionName 'less';
     ?p ?o.
}
```

Listing 3.12: query for the "less_equals" function

3.6 Executing remote queries

In the example we have seen so far, when we needed some georeferenced coordinates, we have provided them manually. One of the main goals of this thesis is to provide a mechanism to integrate raster data and Linked Data.

In a real world application, we want to be able to query some georeferenced points or whole polygons directly from a remote endpoint like dbpedia and link these coordinates somehow to one or more georeferenced rasters to perform calculations. Therefore we have to be able to "outsource" a part of the whole Linked Raster Data query to a remote endpoint.

SPARQL already supports queries on multiple endpoints with federated queries and the SERVICE keyword [1]. In Listing 3.13 is an example that combines the results of a local graph from a file (myfoaf.rdf) with the results of a remote graph with the IRI "http://people.example.org/sparql".

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name
FROM <http://example.org/myfoaf.rdf>
WHERE
{
  <http://example.org/myfoaf/I> foaf:knows ?person .
  SERVICE <http://people.example.org/sparql> {
    ?person foaf:name ?name . }
}
```

Listing 3.13: federated query

At the moment of this writing, rdflib does not implement the SERVICE keyword. But there exists another method to query remote graphs. With the python library "rdflib sparqlstore" [19], we can wrap a remote endpoint into an rdflib graph object and add it as a named graph to our dataset. We then have to make our remote query inside the GRAPH-section of the SPARQL query, because we do not want to query the default graph but the named graph. An example where we obtain the coordinates of five municipalities in the canton of zurich is illustrated in Listing 3.20

```
PREFIX gdal: <http://scharrenbach.net/gdal#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX prop: <http://dbpedia.org/property/>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT * WHERE {
  GRAPH ?g {
    ?r a dbo:Settlement .
    ?r dbo:country <http://dbpedia.org/resource/Switzerland> .
    ?r prop:canton "Zurich"@en .
    ?r geo:lat ?lat .
    ?r geo:long ?long .
   }
}
LIMIT 5
```

Listing 3.14: remote query

As you can see, we do not specify the name of the graph we want to query. Rdflib does this automatically. To this time, we do not know why we can not state the graph IRI explicitly. We have discussed the same issue in Section 3.1.3.

3.7 Integration of WMS services

WMS stands for Web Map Service and is a standard protocol for requesting georeferenced maps from a server [20]. WMS supports different request types. The two most important are:

- "GetCapabilities - returns parameters about the WMS (such as map image format and WMS version compatibility) and the available layers (map bounding box, coordinate reference systems, URI of the data and whether the layer is mostly opaque or not)" [20]
- "GetMap - returns a map image. Parameters include: width and height of the map, coordinate reference system, rendering style, image format" [20]

To get a better idea of a GetMap request, consider the following real world example. The url in Listing 3.15 retrieves a forest map of the canton of zurich in switzerland as a png image.

```
http://wms.zh.ch/WaldarealZHWMS?LAYERS=WaldarealZHWMS
&SERVICE=WMS&VERSION=1.3.0&REQUEST=GetMap
&FORMAT=image%2Fpng%3B%20mode%3D8bit&CRS=EPSG%3A21781
&BBOX=680000,243000,696931,255698&WIDTH=800&HEIGHT=600
```

Listing 3.15: GetMap request for retrieving a categorical raster

When we look closer at the query, we can see that there are different parameters included. In Listing 3.16 is a short description for each of them:

```
service - service name
request - operation name
version - service version
layers - layers to display
srs - Spatial Reference System for map output
format - format for the map output
width - width of map output (in pixels)
height - height of map output (in pixels)
bbox - bounding box for map extent (minx,miny,maxx,maxy)
```

Listing 3.16: Query parameters of a GetMap request [21]

Of course, we do not want to copy and paste a url in the browser window to retrieve the map. We have to find a way to call a wms service in an automated fashion and convert the resulting image (png in this case) to the GeoTiff format, such that we can add it as a new raster graph in our application (see listing 3.19).

3.7.1 Implementation

There are multiple ways to implement an automated WMS service call. We will focus on WMS service calls using the gdal library.

The gdal library provides us with the "gdal_translate" command. With "gdal_translate" we can transform our image result into another file format. We can also change the Spatial Reference System of the result file, if we want.

With "gdal_translate", we have two ways of implementing our service call:

Using a service description file When we use a service description file, we provide the parameters of our service call as an XML document. An example could look like the example in Listing 3.17

```
<GDAL_WMS>
    <Service name="WMS">
        <Version>1.1.1</Version>
        <ServerUrl>http://ganimed.geoportal.dgu.hr/cwms</ServerUrl>
        <SRS>EPSG:3765</SRS>
        <CRS>CRS:HTRS96</CRS>
        <ImageFormat>image/jpeg</ImageFormat>
        <Layers>DOF</Layers>
    </Service>
    <DataWindow>
        <UpperLeftX>399609</UpperLeftX>
        <UpperLeftY>4887306</UpperLeftY>
        <LowerRightX>400326,</LowerRightX>
        <LowerRightY>4888023</LowerRightY>
        <TileLevel>19</TileLevel>
    <TileCountX>1</TileCountX>
    <TileCountY>1</TileCountY>
    </DataWindow>
    <Projection>EPSG:3765</Projection>
    <BlockSizeX>256</BlockSizeX>
    <BlockSizeY>256</BlockSizeY>
    <BandsCount>3</BandsCount>
</GDAL_WMS>
```

Listing 3.17: A service description file [21]

With a service description file, the gdal_translate command could look like the example in Listing 3.18.

```
gdal_translate -of GTiff -outsize 256 256 gdal_wms_dgu.xml geotiff.tif
```

Listing 3.18: gdal_translate command

This approach is not very suitable for us, since we would have to generate an xml file for each request. This can be a large overhead, especially when we have a lot of requests.

Using a GetMap request directly A faster and more practical approach to specify a WMS service call, is to just provide a GetMap request in the query. This way, a user can just copy and paste the url of the map he wants into the query and specify the region he wants as a polygon.

Since we can not pass the GetMap request directly to the gdal_translate command, we have to first get the image result from the server and than use gdal_translate to convert it into the GeoTiff format with a specified Spatial Reference System. This can be done with a simple shell-script like in Listing 3.19

```
ulx="$2"
lrx="$4"
lry="$5"
uly="$3"
crs=EPSG:"$6"
url="$1"
curl -o temp.png $url
gdal_translate -a_srs ${crs} -a_ullr $ulx $uly $lrx $lry temp.png temp.tif
```

Listing 3.19: shell-script for retrieving a remote raster

We pass the Bounding Box and the Spatial Reference System as parameters, download the image and than translate it to GeoTiff.

A query that retrieves a map from a WMS service is illustrated in Listing 3.20.

```
PREFIX gdal: <http://scharrenbach.net/gdal#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX prop: <http://dbpedia.org/property/>
SELECT ?r ?avg WHERE {

  ?r a dbo:Settlement .
  ?r dbo:country <http://dbpedia.org/resource/Switzerland> .
  ?r prop:canton "Zurich"@en .
  ?r geo:lat ?lat .
  ?r geo:long ?long .

  ?poly gdal:Function gdal:CreatePolygonFromPoint.
        gdal:Param1 ?long.
        gdal:Param2 ?lat.
        gdal:Param3 2000.

  ?raster_iri gdal:Function gdal:LoadRasterFromURL.
              gdal:Param1 "http://wms.zh.ch/WaldarealZHWMS?LAYERS=
                  WaldarealZHWMS&SERVICE=WMS&VERSION=1.3.0&REQUEST=GetMap&
                  FORMAT=image%2Fpng%3B%20mode%3D8bit&CRS=EPSG%3A21781&BBOX
                  =680000,243000,696931,255698&WIDTH=800&HEIGHT=600".
              gdal:Param2 "wald".
              gdal:Param3 ?poly.

}
```

Listing 3.20: query for loading a raster remotely

3.8 Applications

To demonstrate the use of Linked Raster Data, one of the main goals of this work is to develop different modules with specific or general use cases. In this thesis, we will implement a small amount of general mathematical functions, map algebra operations and geometric operations to demostrate the possibilities of Linked Raster Data.

Of course there are a huge amount of other operations from different domains we could also implement, but that would take too much time for this thesis and is the subject of future work.

3.8.1 Tomlins Map Algebra

Map Algebra is a set-based algebra. The main idea is to provide a set of primitive mathematical operations, which take two or more raster layers (maps) as input and return a new raster layer as output.

The provided operations include all primitive binary operations, such as addition, subtraction, multiplication and division. Boolean operations can also be implemented. In general, each operation that takes multiple maps as input and returns a new map can be a map algebra operation.

In a further distinction, the operations can be categorized into 4 classes: local, focal, global, and zonal. Local operations operate on single raster cells, or pixels. Focal operations operate on cells and their surrounding neighbors, whereas global operations operate on the entire map. Finally, zonal operations operate on areas of cells that have the same value [22].

As being said, the output of each operation is a itself a map, therefore, the operations can be combined into a procedure or script, in order to perform more complex tasks. [22]

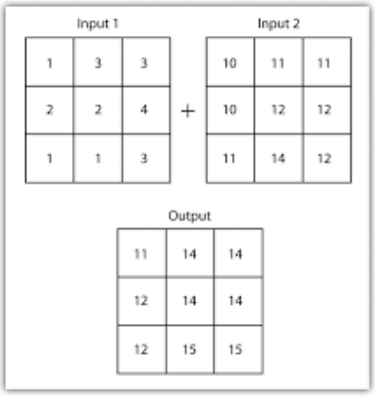


Figure 3.4: Addition of 2 rasters [23]

In our application, we implemented the following Map Algebra operations:

- Addition of two raster

- Subtraction of two raster
- Multiplication of two raster
- Division of two raster
- Logical AND operation on two raster
- Logical OR operation on two raster
- Logical NOT operation on two raster
- Smaller operation on a raster
- Smaller-equals operation on a raster
- Greater-equals operation on a raster
- Equals operation on a raster
- Not-equals operation on a raster

3.8.2 Basic mathematical operations

In addition to our map algebra opertions, we need a small amount of aggregation functions that return only one value per raster. This is useful, when we want to have information about the values of a band as a whole.

One could argue that these are also map algebra operations since a number can be seen as a 1x1 matrix. However, in this thesis we decided to separate all aggregation functions from the map algebra operations, because it doesnt make sense to generate a raster out of a single value.

We implemented the following operations:

- Minimum value
- Maximum value
- mean value
- median value

3.8.3 Geometric operations

When we want to define regions for which we want to perform raster operations, we often have multiple polygons to intersect. Suppose we have the coordinates of two municipalities and we want to know the percentage of forest that covers the intersection of the two places in a radius of four kilometers.

To answer this query we have to first generate a polygon for all places that covers the area in a radius of four kilometers. Then we have to intersect those polygons to obtain

the final region for which we have to aggregate a map containing the forest area of these regions.

For obtaining the two polygons only from the coordinates of the places and a desired radius, we implemented a function "GeneratePolygonFromPoint" which takes the longitude and latitude of a georeferenced point as well as a radius in meters to generate a polygon with four points that lay on the circle defined by the radius.

There are already GeoSPARQL endpoints that support geometric operations on polygons, for example for retrieving all places that lay in a certain area. However in our endpoint we have to implement these operations ourselfs. For demonstration purposes, we implemented the following operations:

- Intersection of 2 polygons
- Difference of 2 polygons
- Union of 2 polygons

4

Evaluation

4.1 Setup

As we saw in Chapter 2, we can evaluate our query for obtaining the geocoordinates on which we want to calculate either locally or from a remote endpoint such as dbpedia.

In this chapter, we will measure the execution time with respect to different queries (usecases) and different scenarios. The usecases differ in the number of raster files on which we calculate our results. The resolution of the clipped regions stays always constant. The scenarios differ in the way we fetch the rasters and geo-coordinates (either local or remote). Another variable for the scenarios is the number of places, for which will calculate the results.

The 3 usecases are formally defined as follows:

1. In the first usecase (UC1), we will use one raster. We make one projection (logical less operation) and one aggregation (average) on the raster for each place.

2. In UC2, we will use two raster. We make three projections (2 x less operation, 1 x logical-and) and one aggregation (average) on the raster for each place.

3. In UC3, we will use three raster. We make five projections (3 x less operation, 2 x logical-and) and one aggregation (average) on the raster for each place.

Each usecase is executed in four different configurations:

1. The fetching of the raster files as well as the query for the geo coordinates are executed remote. That means, we fetch the rasters via WMS and the geo coordinates via dbpedia.

2. The fetching of the raster files as well as the query for the geo coordinates are executed locally. That means, we load the rasters from a local file and we query the local default graph for the coordinates.

3. We fetch the raster files via WMS and the geo coordinates locally.

4. We load the raster files locally and the geo coordinates via dbpedia.

With this configurations, we are able to measure the difference between local and remote execution. Further we can examine different query execution strategies when executing locally or remotely. As a third variable, we will execute each configuration for one and five places on the map. This way, we will find out, how the query scales with respect to the number of results.

Each scenario is executed 20 times. Scenarios with remote queries are executed during different day times, since we dont want biased results resulting from different loads on the server.

4.1.1 The three usecases

In the following subsection, we will look at the three usecases in more detail. All three rasters are categorical rasters of the canton of zurich that are available online on the WMS server [24]. The places are all municipalities that lay also in the canton of Zurich. We will only give an abstract description of each query, since the actual queries are very long. You can find them in Appendix A.1.

UC1 In the first query, we obtain the coordinates of a number of municipalities and calculate for each the percentage of forest area in a radius of two kilometers. We need a map with the forest area in the canton of zurich and mark all values that are not 255 (white) with a one. We than calculate the average of the resulting map to obtain the percentage of forest in this area.

The query of UC1 is outlined in Listing 4.1.

```
SELECT ?avg WHERE {

 Match coordinates.
 create a polygon from the coordinates.
 Clip the raster.
 Match the first band.

 GRAPH ?g{
   Retrieve pixelvalues (pv) for the band.
 }

 execute function "smaller" (smaller) on pv.
 execute function "average" for smaller.

}
```

Listing 4.1: query of UC1. For the full query see A.1

UC2 In the second query, we retrieve the coordinates of a number of municipalities and calculate for each the percentage of forest area that lays on archelogical sites in a radius of two kilometers. In addition to the forest map, we also make a projection on the archeological map to mark the archelogical sites. We than calculate the `logical_and` function to get the intersection of forest sites and archeological sites. At the end we obtain the average value of the resulting map like we did before.

The query of UC2 is outlined in Listing 4.2.

```
SELECT ?avg WHERE {

 Match coordinates.
 create a polygon from the coordinates.
 Clip the first raster.
 Clip the second raster.
 Match the first band of raster1.

 GRAPH ?g{
   Retrieve pixelvalues (pv1) for the band.
 }

 Match the first band of raster2.

 GRAPH ?g{
   Retrieve pixelvalues (pv2) for the band.
 }

 execute function "smaller"(smaller1) on pv1.
 execute function "smaller"(smaller2) on pv2.
 execute function "logical_and" (proj) on smaller1 and smaller2
 execute function "average" for proj.

}
```

Listing 4.2: query of UC2. For the full query see A.2

UC3 In the last query, we add a third raster that contains all memorial places in the canton of zurich. As before, we additionally project the third raster to a binary map, calulate the intersection to the other 2 maps and take the average as result.

The query of UC3 is outlined in Listing 4.3

```
SELECT ?avg WHERE {

 Match coordinates.
 create a polygon from the coordinates.
 Clip the first raster.
 Clip the second raster.
 Clip the third raster.
 Match the first band of raster1.

 GRAPH ?g{
   Retrieve pixelvalues (pv1) for the band.
 }

 Match the first band of raster2.

 GRAPH ?g{
   Retrieve pixelvalues (pv2) for the band.
 }

 Match the first band of raster3.

 GRAPH ?g{
   Retrieve pixelvalues (pv3) for the band.
 }

 execute function "smaller"(smaller1) on pv1.
 execute function "smaller"(smaller2) on pv2.
 execute function "smaller"(smaller3) on pv3.
 execute function "logical_and" (proj) on smaller1 and smaller2.
 execute function "logical_and" (proj2) on smaller3 and proj.
 execute function "average" for proj2.

}
```

Listing 4.3: query of UC3. For the full query see A.3

4.2 Results

In this section we will illustrate the results of our experiments. In table 4.2, we summarized the means and standard deviations of all evaluation runs. In Appendix A.2.1 you will also find the plots for all configurations.

#rasters	raster	places	#places	mean	std
1	remote	remote	1	2.44s	1.45s
1	local	local	1	0.15s	0.01s
1	remote	local	1	0.44s	0.07s
1	local	remote	1	2.00s	0.77s
1	remote	remote	5	36.05s	18.71s
1	local	local	5	0.35s	0.02s
1	remote	local	5	0.61s	0.03s
1	local	remote	5	35.52s	7.20s
2	remote	remote	1	7.28s	2.66s
2	local	local	1	0.30s	0.03s
2	remote	local	1	0.77s	0.03s
2	local	remote	1	1.99s	0.27s
2	remote	remote	5	46.33s	15.77s
2	local	local	5	0.69s	0.02s
2	remote	local	5	1.13s	0.05s
2	local	remote	5	39.27s	11.27s
3	remote	remote	1	4.37s	1.92s
3	local	local	1	0.46s	0.02s
3	remote	local	1	1.18s	0.14s
3	local	remote	1	2.98s	1.09s
3	remote	remote	5	43.34s	12.80s
3	local	local	5	1.03s	0.02s
3	remote	local	5	1.68s	0.05s
3	local	remote	5	39.35s	8.91s

4.3 Observations

When we look at the different statistics in the results section, we can immediately see some big differences in the execution time of the different scenarios:

- All queries that evaluate the coordinates via remote endpoint, take much longer. This observation is independent of the usecase. This is what we would expect, since it takes time to query the endpoint and the load of the server can differ significantly over time.
- All queries that evaluate the coordinates via remote endpoint have a relatively large variance in comparison to local queries. This is also because the load of the server can vary over time.
- The overhead for fetching the raster via WMS is small and constant. This is most probably because this particular WMS server is visited less frequently than for example the server of dbpedia. For other servers, this may not be the case and the overhead can be significantly larger.

- In all queries that obtain the places remote, the execution time grows overproportional to the number of places. This means that the more results we have to fetch from the remote endpoint, the larger the overhead will be.

Implications on the query structure As mentioned above, the number of results that we fetch from a remote endpoint can have a big impact on the query execution time. Therefore it is important to filter the results on the remote endpoint and not locally. This is also important since local FILTER expressions are evaluated after all other expressions (see Section 3.1.3). This means that we would execute operations on the rasters for results we actually do not want.

5

Discussion

Based on our goals we have defined in the introduction, we can state the following:

To connect the data sets with existing Linked Open Data (LOD), such as GeoNames and dbPedia in order to link entities from the Linked Open Data into the raster data.

- In the GeoSPARQL examples of the introduction chapter, we saw a way of linking vector data to other linked data resources. Our goal was to implement similar functions for linking raster data to linked data resources and express them as a set of Basic Graph Patterns.

  In Section 3.5 we saw one possible approach of describing our custom functions in RDF. We saw how we can introduce our own predicates and resources for the meta data and using them to describe our functions.

  In Section 3.1.3 we have explained that the Basic Graph Pattern approach can not be implemented in rdflib, because of the execution order of the query engine. Therefore, we had to find a workaround for making our function calls. We saw, that we can achieve the same functionality with help of extension functions and BIND expressions.

  An example of the original approach we saw in Listing 3.2. An example of the second approach we saw in Listing 3.5.

  In order to register our custom functions in the query engine, we had to modify rdflib. We also have to take care when mixing BGPs and extension functions because of the query execution order in rdflib.

  In Section 3.6 we saw that we can not use federated queries to query remote endpoints since rdflib does not implement the SERVICE keyword. We have found a workaround by using the rdflib-sparqlstore [19] wrapper for the dbpedia endpoint and add it as a named graph to our dataset. This way, we are able to execute remote queries on the dbpedia endpoint. However, to this time, we can not specify the endpoint we want to query because rdflib does not allow that. Therefore we can only query one endpoint at the moment.

  In Chapter 4, we saw that it is important to filter the results on the remote server and not locally, because otherwise, we will compute all raster operations for all results and the query will take significantly longer.

To assemble statistical data, preferably from Swiss municipal, cantonal and federal offices, and link these to Linked Open Data and the raster data.

- We did not succeed in assembling enough data from offical sources to achieve this goal. Instead, we decided to implement WMS service calls to load raster files from remote sources.

  In Section 3.7 we have discussed two approaches to implement WMS service calls in our application:

  – Using a service description file to specify the call and retrieving the image result and using `gdal_translate` to transform into the GeoTiff format.

  – Using a GetMap request directly to retrieve the image result from the server and using `gdal_translate` to transform it into the GeoTiff format.

  We have concluded that the second approach is more suitable for our needs since we dont have to generate an XML file for each request and because the user has to specify less parameters but can just copy the url into the SPARQL query.

  In Subsection 5.1.1 we have discussed that it would be desirable to load rasters without explicitly copying the GetMap request into the query. Instead we could use an index server to retrieve rasters by category. We explained that this is not yet possible because the following conditions have to be fullfilled:

  – We need a (or very few) servers for every country to retrieve the metadata for all rasters.

  – We need a protocol (or extend the WMS protocol) that supports categorization of rasters.

To assemble a dataset of raster data, preferably of Swiss municipal, cantonal and federal resources.

- Since we implemented WMS service calls in our application, we used the WMS server of the canton of zurich [24] to retrieve the rasters for our examples and usecases.

In the following section, we will discuss the limitations of our application so far and what future work needs to be done in order to fully integrate raster data with Linked Data in a real world application.

5.1 Limitations

Altough we worked out a possible query structure and some additional features useful for Linked Raster Data queries, we still have limitations in our application:

- the lack of a broad support of raster functions for different domains and raster types. We will discuss this issue in Section 5.1.2.

- the limited support for WMS services we have implemented so far. We will discuss how we could improve the support in Section 5.1.1.
- at the moment, our application does only support one remote endpoint (dbpedia). We will discuss how to add additional endpoints in Section 5.1.3.

5.1.1 Indexing of rasters

So far, the user of our enpoint can load a raster from a WMS server by copying the GetMap-request for the map he needs directly into the query (see Section 3.7). It also can restrict the result map to the polygon he provides to our function.

This method described above is only suitable when the user of our endpoint knows exactly which raster he needs and where to find it. It is also possible, that the user does not know where to search for WMS services or he does not know which raster he will need for his query.

It would be very useful when the user could search for rasters by category and/or location that he needs for his query and our server would automatically retrieve one or multiple rasters that match his conditions.

One possibility to make this possible, is to implement an index server for WMS requests, that stores the metadata for every GetMap request:

- the boundingbox of the area the raster covers (in geo coordinates)
- the Spatial Reference System of the raster
- the category/categories of the data the raster stores
- the GetMap request to retrieve the raster

With this information, we are able to resolve maps for which the user has only to specify the category and a point or polygon that has to lay in the raster. A further restriction could be the number of maps he wants to use, since there can be multiple maps satisfying his conditions. He could also select the average value from a specified number of rasters. This behaviour could be useful to correct outliers or inaccurate results.

Suppose a user wants to query a map containing elevation data for the point (7.566, 47.695). He could then load a suitable map with an expression like in Listing 5.1.

```
?raster gdal:Function gdal:LoadRasterByCategory.
        gdal:Param1 "elevation".
        gdal:Param2 POINT(7.566 47.695).
```

Listing 5.1: A possible query for loading a raster by category

In this simple example, we assume that we take the first raster that satisfies our condition.

Challenges In order to implement such an index server, a lot of work needs to be done. For every country, there exist a huge amount of rasters that are usually not available from a central WMS server. If there was a central WMS server for every country, we could make a GetCapabilities request (see Section 3.7) to retrieve most metadata for a map like the boundingbox, the supported Spatial Reference Systems and supported image formats. But there are no raster categories available in WMS and therefore we would still have to look at the data and decide for every map in which category it belongs. Doing this for every country would still need a lot of workforce and time.

So we can conclude that to make a raster index server possible on a global scale, there are two main challenges that have to be done:

- We need a (or very few) servers for every country to retrieve the metadata for all rasters.
- We need a protocol (or extend the wms protocol) that supports categorization of rasters.

5.1.2 Implementation of additional functions

So far we saw a small number of logical and mathematical functions in Section 3.8. However there are still a lot of possible operations that could be implemented.

Consider for example that you want to go on a ski tour and you want to know where the avalanche zones lay. Avalanche zones are zones with an ascending slope greater than 30 degrees. When we dont have a map containing the slopes of an area but only the elevation map, we would need to calculate the derivative for our map.

Another example could be the statistical evaluation of multiple maps of the same category that cover the same area. A user could be interested in comparing different statistical properties of the maps. In this case we would need to implement further statistical functions for the calculation of the variance or the percentiles of the dataset.

A limitation that we have not discussed yet, is the interpretation of categorical raster data and its implications on our customfunctions. Consider for example that we have a categorical raster containing the population density of a country and we want to know the number of people that live in a an area bounded by a polygon.

In this case we have to first resolve the actual densities for every pixel in the region, since we can not work directly with the colour value of the map. We can obtain the density value of a pixel with help of the metadata that is encoded in the raster file. We than perform a weighted sum over all pixels in the bounded area. The weight of a pixel is its area in square kilometers.

Aggregation operations on categorical rasters are at the moment only partially possible, because we would need to implement special functions for density rasters such as the transformation into actual densities instead of colour avalues and the calculation of the weighted sum.

5.1.3 Support for more remote endpoints

At the moment, we can only query the remote endpoint of dbpedia (see Section 3.6). In a real world application it would be desirable to specify different endpoints in our query, since a lot of ressources and functions are not available on every endpoint.

Since rdflib does not support the "SERVICE" keyword and we can not specify the named graph we want to query (see also Section 3.6), this is not possible at the moment. We hope that in the future, the support for federated queries in rdflib will be implemented.

6

Related Work

6.1 Geo Data as Linked Data

Fleischli [5] wrote a master thesis about the topic of exposing geo data as Linked Data. He discussed standards for linking vector data, such as GeoSPARQL [**?**] or linkedgeodata.org [2] as well as new approaches for the linking of raster data.

In his thesis he discusses two possible approaches to link raster data:

- *The VKE approach*

  The VKE approach stands for "Vektorisierte Kleinste Einheiten" and can be used to represent raster as well as vector data in RDF. The idea is to extract features from a raster and store them explicitly in RDF such that they can be queried directly without further processing. This is the same concept as the "explicit features" approach we discussed in Section 2.3.

- *The ARO approach*

  ARO stands for "Ausgegliederte Räumliche Operationen". It is the idea of using parametrized geo operations to calculate features of a raster in a context sensitive way. It is the same idea like the "feature description" approach we discussed in Section 2.3.

In his thesis, he concludes that the first approach is difficult to implement since the implicit fatures in the raster have to be explicitly stored in RDF. This means that we have to decide which features we want to extract in what quality. Using this approach, we are not able to answer all possible usecases for a raster. We have discussed the same problem in Section 2.3 and came to the same conclusion.

6.2 Linked Raster Data

Scharrenbach, Bischof, Fleischli and Weibel [4] also discussed the theory and possible imlementations of Linked Raster Data in their paper. Concerning the implementation of Linked Raster Data, they came up with the two approaches of **explicit features** and **feature descriptions**, which we have already discussed in Section 2.3.

The paper also proposes to use RDF to describe the semantics and parameters of geo-operations, such that they can be queried on the endpoint. We discussed the description of functions in Section 3.5.

7

Conclusions

In chapter 1 of this document, we discussed two approaches of implementing Linked Raster Data: explicit features and feature description. We decided to implement the features description approach.

In Chapter 3, we saw a possible implementation of an endpoint. We discussed a possible query structure and its implementation in python with help of rdflib and lrdgdal. We did not succeed in implementing the Basic Graph Pattern approach in our application, because of the execution order of the query engine. We had to find a workaround for making our function calls. We simulated the same functionality with help of extension functions and BIND expressions. We conclude that we were successfull in linking Linked Data to raster data but the implementation is not optimal because of the lack of certain features in the SPARQL engine of rdflib.

Further, we saw how to integrate remote endpoints into our queries in Section 3.6. Since rdflib does not implement the SERVICE keyword, we had to find a workaround by using the rdflib-sparqlstore wrapper for the endpoint. With this implementation, we can not query more than one remote endpoint, because we can not explicitly define the graph, on which the query is executed. We conclude that the integration of remote endpoints is not a problem when the SPARQL engine supports federated queries.

We were not able to collect a lot of raster data from official sources. Instead, we decided to implement WMS service calls to load raster files remotely and use a WMS server to obtain the maps for our queries. We conclude that using the GetMap request directly in our queries to specify the service call, is the most suitable approach for us, because we do not need to specify a service description file for every call. In Section [**?**], we conclude that in order to search rasters by category, we would need a central map server for every country as well as a protocol, that supports raster categories.

In Chapter 4, we evaluated three usecases in different configurations. We conclude that remote queries take significantly longer. Further we conclude that it is important to filter the results on the remote server and not locally, because otherwise, we have to compute all raster operations for all results.

Finally, we can conclude that we successfully implemented an approach that combines Linked Data and raster data with help of the SPARQL query language. We hope that in the future, there will be more research in the area of Linked Raster Data.

References

[1] Antoniou, van Harmelen *A semantic Web primer.* Cambridge, Massachusets, MIT Press, 2008

[2] http://linkedgeodata.org/About

[3] http://openstreetmap.org

[4] Scharrenbach, Bischof, Fleischli, Weibel. *Linked Raster Data.* University of Zurich, Department of Informatics, Zurich

[5] Fleischli (2012). *Geodaten als Linked Data Eine Untersuchung zur Strukturierung und Vernetzung von Umweltdaten für das Semantic Web.* Geographic Institut, University of Zurich, Zurich

[6] http://en.wikipedia.org/wiki/Map_projection

[7] http://en.wikipedia.org/wiki/File:Usgs_map_miller_cylindrical.PNG

[8] http://en.wikipedia.org/wiki/File:Usgs_map_azimuthal_equidistant.PNG

[9] http://en.wikipedia.org/wiki/Spatial_reference_system

[10] http://www.gdal.org/gdal_datamodel.html

[11] http://www.osgeo.org/

[12] http://www.gdal.org

[13] https://pypi.python.org/pypi/lrdgdal/0.0.4

[14] http://www.rdflib.net

[15] http://resources.esri.com/help/9.3/arcgisengine/java/gp_toolref/geoprocessing/GPKAC_Extract_by_mask.g

[16] https://trac.osgeo.org/mapguide/attachment/wiki/MapGuideRfc51/MapRasterIntersection.png

[17] http://docs.scipy.org/doc/scipy-0.13.0/reference/generated/scipy.ndimage.interpolation.zoom.html

[18] http://resources.arcgis.com/en/help/getting-started/articles/GUID-5BD98316-8BF4-430C-BA24-F47EFF8BF66D-web.png

[19] https://github.com/RDFLib/rdflib-sparqlstore

[20] http://en.wikipedia.org/wiki/Web_Map_Service

[21] http://ipasic.com/article/get-data-wms-layers-using-gdal-and-python/

[22] https://en.wikipedia.org/wiki/Map_algebra

[23] http://2012books.lardbucket.org/books/geographic-information-system-basics/section_12/095c6533afae0ad16c99688a3d0d489e.jpg

[24] http://www.geolion.zh.ch/opendata

[25] https://en.wikipedia.org/wiki/GeoSPARQL

A

Appendix

A.1 Raster queries from Chapter 4

```
PREFIX gdal: <http://scharrenbach.net/gdal#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX prop: <http://dbpedia.org/property/>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?avg WHERE {

  GRAPH ?g{

    ?r a dbo:Settlement .
    ?r dbo:country <http://dbpedia.org/resource/Switzerland> .
    ?r prop:canton "Zurich"@en .
    ?r geo:lat ?lat .
    ?r geo:long ?long .

  }

  ?poly gdal:Function gdal:CreatePolygonFromPoint.
        gdal:Param1 ?long.
        gdal:Param2 ?lat.
        gdal:Param3 2000.

  ?raster gdal:Function gdal:ClipRasterToPolygon.
          gdal:Param1 "http://example.com/raster".
          gdal:Param2 ?poly.
          gdal:Param3 "wald".

  ?raster gdal:Band ?band.
  ?band gdal:BandNumber 1.

  GRAPH ?band{
     ?pv1 gdal:Function gdal:GetPixelValues.
  }

  ?less gdal.Function gdal:smaller.
        gdal:Param1 ?pv1.
```

```
        gdal:Param2 255.

  ?avg gdal:Function gdal:average.
       gdal:Param1 ?less.
}
```

Listing A.1: query of the first usecase

```
PREFIX gdal: <http://scharrenbach.net/gdal#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX prop: <http://dbpedia.org/property/>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?avg WHERE {

  GRAPH ?g{

   ?r a dbo:Settlement .
   ?r dbo:country <http://dbpedia.org/resource/Switzerland> .
   ?r prop:canton "Zurich"@en .
   ?r geo:lat ?lat .
   ?r geo:long ?long .

  }

  ?poly gdal:Function gdal:CreatePolygonFromPoint.
        gdal:Param1 ?long.
        gdal:Param2 ?lat.
        gdal:Param3 2000.

  ?wald gdal:Function gdal:ClipRasterToPolygon.
        gdal:Param1 "http://example.com/raster1".
        gdal:Param2 ?poly.
        gdal:Param3 "wald".

  ?archeo gdal:Function gdal:ClipRasterToPolygon.
          gdal:Param1 "http://example.com/raster2".
          gdal:Param2 ?poly.
          gdal:Param3 "archeo".

  ?raster1 gdal:Band ?band1.
  ?band1 gdal:BandNumber 1.

  GRAPH ?band1{
     ?pv1 gdal:Function gdal:GetPixelValues.
  }

  ?raster2 gdal:Band ?band2.
  ?band2 gdal:BandNumber 1.

  GRAPH ?band2{
     ?pv2 gdal:Function gdal:GetPixelValues.
  }
```

```
  ?less1 gdal.Function gdal:smaller.
         gdal:Param1 ?pv1.
         gdal:Param2 255.

  ?less2 gdal.Function gdal:smaller.
         gdal:Param1 ?pv2.
         gdal:Param2 255.

  ?proj gdal:Function gdal:logical_and.
        gdal:Param1 ?less1.
        gdal:Param2 ?less2.

  ?avg gdal:Function gdal:average.
       gdal:Param1 ?proj.
}
```

Listing A.2: query of the second usecase

```
PREFIX gdal: <http://scharrenbach.net/gdal#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX prop: <http://dbpedia.org/property/>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?avg WHERE {

  GRAPH ?g{

   ?r a dbo:Settlement .
   ?r dbo:country <http://dbpedia.org/resource/Switzerland> .
   ?r prop:canton "Zurich"@en .
   ?r geo:lat ?lat .
   ?r geo:long ?long .

  }

  ?poly gdal:Function gdal:CreatePolygonFromPoint.
        gdal:Param1 ?long.
        gdal:Param2 ?lat.
        gdal:Param3 2000.

  ?wald gdal:Function gdal:ClipRasterToPolygon.
        gdal:Param1 "http://example.com/raster1".
        gdal:Param2 ?poly.
        gdal:Param3 "wald".

  ?archeo gdal:Function gdal:ClipRasterToPolygon.
          gdal:Param1 "http://example.com/raster2".
          gdal:Param2 ?poly.
          gdal:Param3 "archeo".

  ?denkmal gdal:Function gdal:ClipRasterToPolygon.
           gdal:Param1 "http://example.com/raster3".
```

```
            gdal:Param2 ?poly.
            gdal:Param3 "denkmal".

  ?raster1 gdal:Band ?band1.
  ?band1 gdal:BandNumber 1.

  GRAPH ?band1{
     ?pv1 gdal:Function gdal:GetPixelValues.
  }

  ?raster2 gdal:Band ?band2.
  ?band2 gdal:BandNumber 1.

  GRAPH ?band2{
     ?pv2 gdal:Function gdal:GetPixelValues.
  }

  ?raster3 gdal:Band ?band3.
  ?band3 gdal:BandNumber 3.

  GRAPH ?band3{
     ?pv3 gdal:Function gdal:GetPixelValues.
  }

  ?less1 gdal.Function gdal:smaller.
         gdal:Param1 ?pv1.
         gdal:Param2 255.

  ?less2 gdal.Function gdal:smaller.
         gdal:Param1 ?pv2.
         gdal:Param2 255.

  ?less3 gdal.Function gdal:smaller.
         gdal:Param1 ?pv3.
         gdal:Param2 255.

  ?proj gdal:Function gdal:logical_and.
        gdal:Param1 ?less1.
        gdal:Param2 ?less2.

  ?proj2 gdal:Function gdal:logical_and.
         gdal:Param1 ?proj.
         gdal:Param2 ?less3.

  ?avg gdal:Function gdal:average.
       gdal:Param1 ?proj2.
}
```

Listing A.3: query of the third usecase

A.2 Plots of the evaluation

A.2.1 All queries of the first usecase

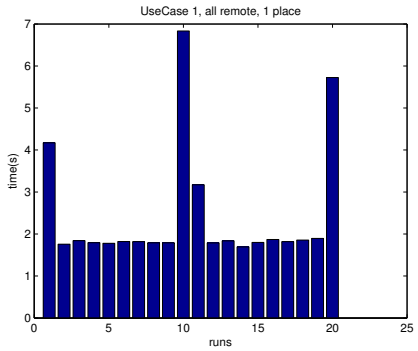


Figure A.1: All remote, 1 Place

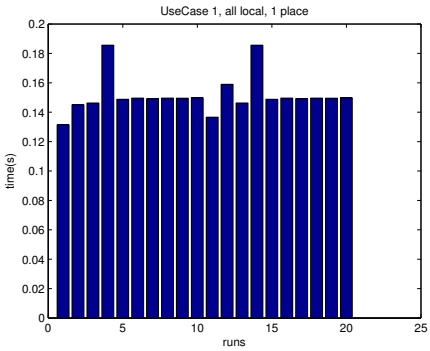


Figure A.2: All local, 1 Place

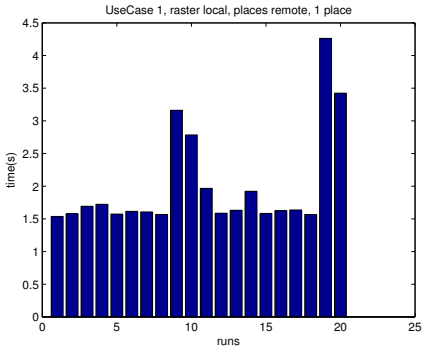


Figure A.3: Raster local, places remote, 1 place

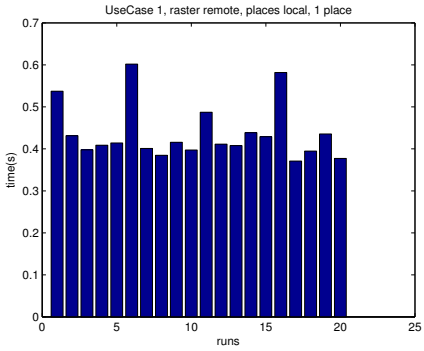


Figure A.4: Raster remote, places local, 1 Place

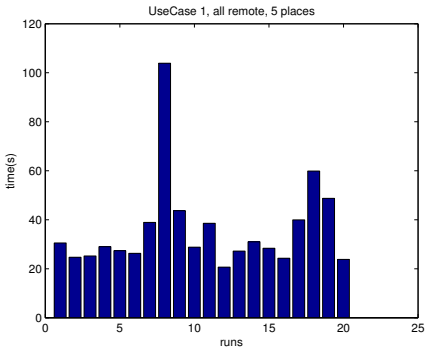


Figure A.5: All remote, 5 Places

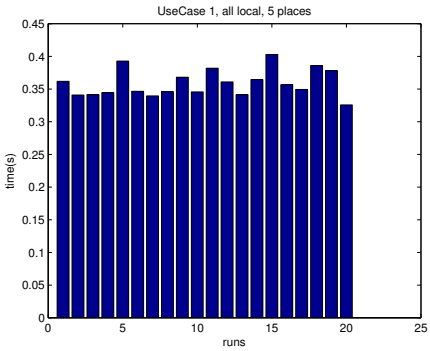


Figure A.6: All local, 5 places

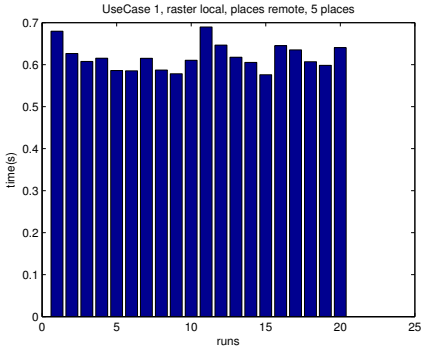


Figure A.7: Raster local, places remote, 5 Places

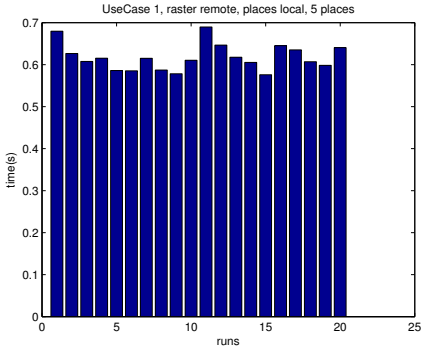


Figure A.8: Raster remote, places local, 5 places

A.2.2 All queries of the second usecase

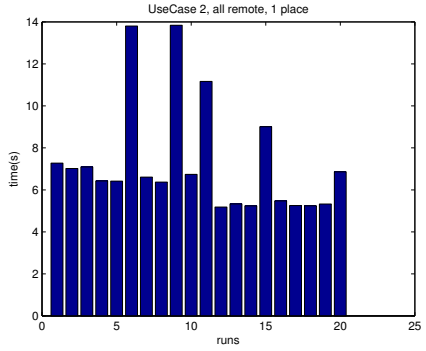


Figure A.9: All remote, 1 Place

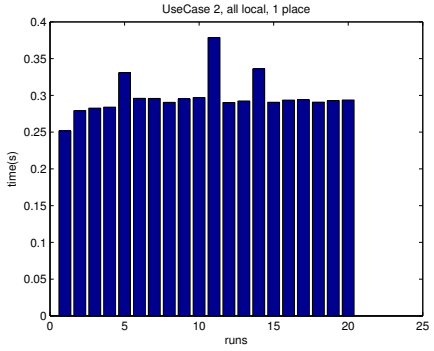


Figure A.10: All local, 1 Place

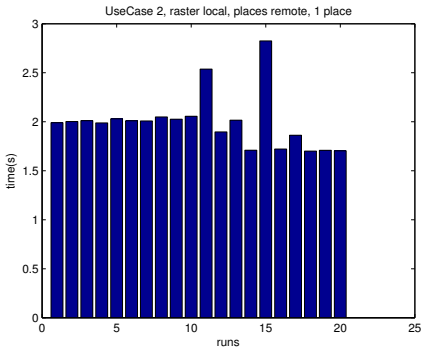


Figure A.11: Raster local, places remote, 1 place

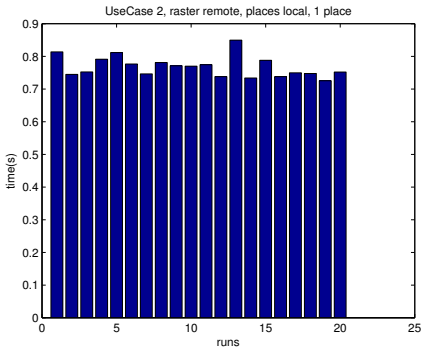


Figure A.12: Raster remote, places local, 1 Place

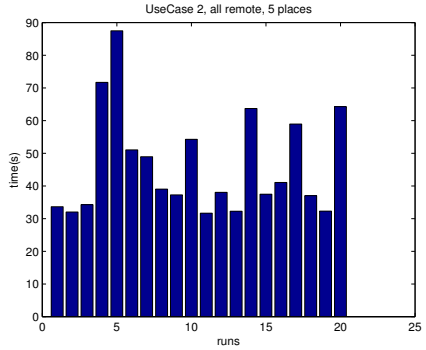


Figure A.13: All remote, 5 Places

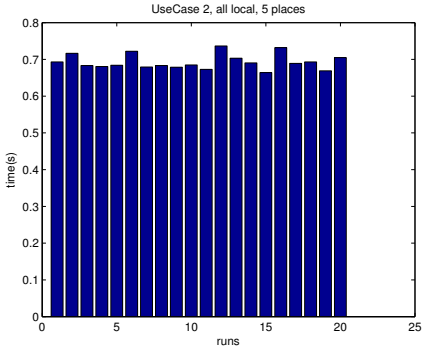


Figure A.14: All local, 5 places

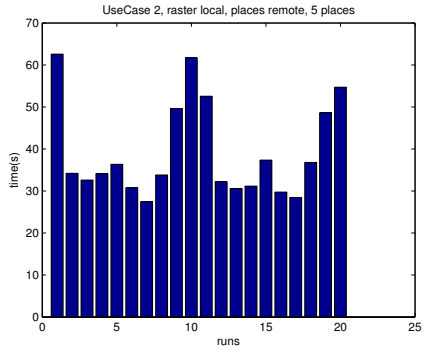


Figure A.15: Raster local, places remote, 5 Places

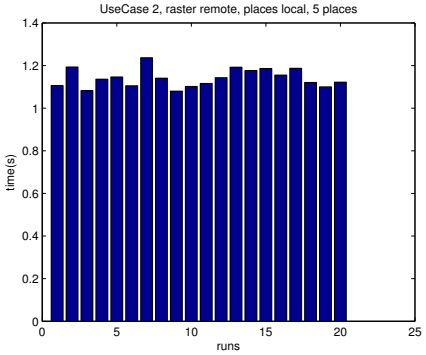


Figure A.16: Raster remote, places local, 5 places

A.2.3 All queries of the third usecase

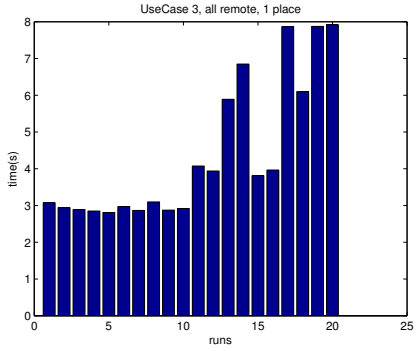


Figure A.17: All remote, 1 Place

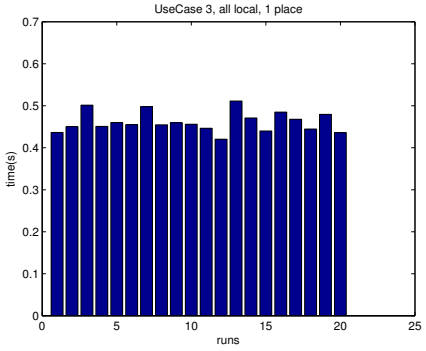


Figure A.18: All local, 1 Place

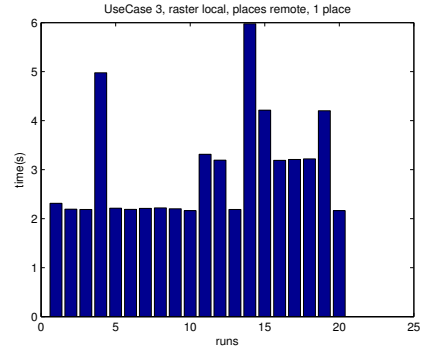


Figure A.19: Raster local, places remote, 1 place

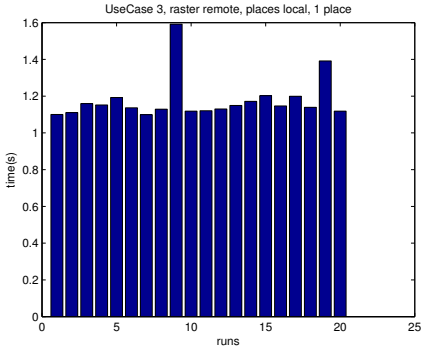


Figure A.20: Raster remote, places local, 1 Place

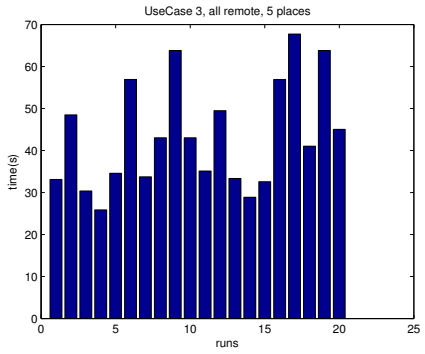


Figure A.21: All remote, 5 Places

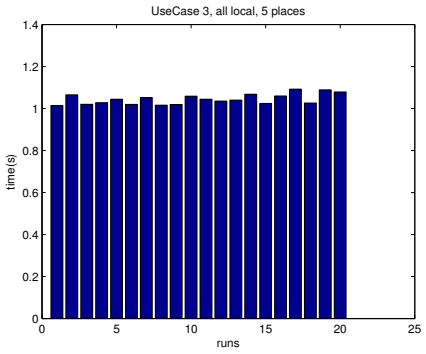


Figure A.22: All local, 5 places

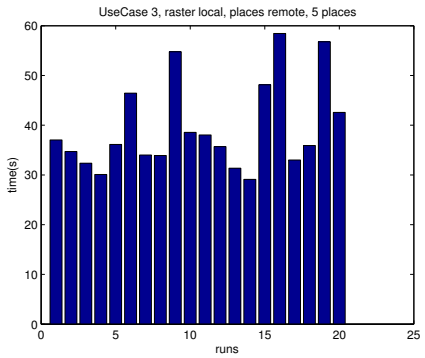


Figure A.23: Raster local, places remote, 5 Places

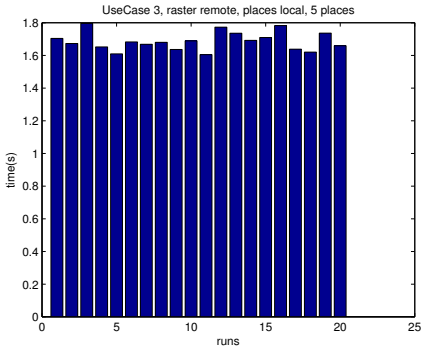


Figure A.24: Raster remote, places local, 5 places

List of Figures

List of Tables