Bachelor Thesis

Efficient Presentation of GeoSPARQL-Results

Denis Veil

Examiner: Prof. Dr. Hannah Bast
Advisers: Patrick Brosi

University of Freiburg

Faculty of Engineering

Department of Computer Science

Chair for Algorithms and Data Structures

September 08th, 2021

Writing Period

08. 06. 2021 – 08. 09. 2021

Examiner

Prof. Dr. Hannah Bast

Advisers

Patrick Brosi

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare that my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Malterdingen, 08.09.2021

Place, Date	Signature

Abstract

QLever is a SPARQL engine, which can handle large datasets [1]. The engine should now be expanded to include geo-functionalities. In doing so, *QLever* returns results for certain queries that contain geometric data.

This thesis describes a web application that is able to send SPARQL queries to the *QLever* engine via an API, then process large results and display these on a map.

The result size of a query can exceed several million. With each entry containing potentially multiple data points the web browser struggles to render them at the same time. The main challenge of the application is to handle even larger results efficiently to deliver a user-friendly experience.

Zusammenfassung

QLever ist eine SPARQL-Engine, die mit großen Datensätzen umgehen kann [1]. Die Engine soll nun um Geo-Funktionalitäten erweitert werden. Dabei liefert *QLever* bei bestimmten Abfragen Antworten zurück, die geometrische Daten beinhalten.

Diese Arbeit beschreibt eine Web-Applikation, die in der Lage ist SPARQL Abfragen an die *QLever*-Engine via einer API zu schicken, dann große Ergebnisse zu verarbeiten und diese auf einer Karte darzustellen.

Die Ergebnisgröße einer Abfrage kann dabei mehrere Millionen überschreiten. Da jeder Eintrag potenziell mehrere Datenpunkte enthält, hat der Webbrowser Schwierigkeiten, sie gleichzeitig darzustellen. Die größte Herausforderung der Anwendung besteht darin, größere Ergebnisse effizient zu verarbeiten, um ein benutzerfreundliches Erlebnis bieten zu können.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

QLever is a SPARQL engine for efficient combined search on a knowledge base and text corpus [1]. Currently, the engine is being fed with datasets, inter alia, generated with OpenStreetMap (OSM) and is now being extended with geo-funcionalities. Certain SPARQL queries return a response with geometric data. This data can contain for example places, streets or buildings in the form of geometric primitives as points, linestrings or polygons, respectively (see Chapter 3).

The size difference in the results of the queries can be enormous. For example, the query for all buildings in Freiburg (Listing 3.1) returns only `298` objects. In comparison, the query for all residential highways (Listing 4.1) returns `57.992.788` objects. We also have to consider that with each entry containing potentially multiple data points the web browser struggles to render them at the same time.

To display this data the engine needs a web frontend that can handle this geometric data efficiently.

1.1 Problem

While it is not a big challenge for modern browsers to handle a few hundred data points, the result size of some queries can reach a mark in the hundreds of thousands or millions. Despite that a point or, respectively, a coordinate takes only about 20 Bytes, a linestring can contain multiple points and a polygon multiple linestrings or even polygons itself. These are being called multipolygons. Therefore the task for browsers to display this geometric data can grow very large and depending on the shape types non-linear.

1.2 Motivation

The focus of this thesis is to develop a powerful web frontend application as an extension to the *QLever* engine for displaying a query result without sacrificing the user experience in terms of loading times or slowdowns.

To approach the large amounts of data we used the open-source JavaScript library *Leaflet* for mapping [2]. The web application is able to read a SPARQL query and obtain the query result over a GET request. The subsequent process decides then based on the extent of the data how to deal with the result. If the extent is too large, the query gets in the first step a filter option to limit the visible geometric shapes. Then the result is being clustered and finally add to the map. Afterwards, only the explored map or place of interest is being loaded in the background.

2 Related Work

2.1 Query Language

SPARQL, short for *SPARQL Protocol And RDF Query Language* [3], is a query language and protocol for retrieving RDF data [4]. The RDF model is represented as a directed graph which contains triples. A triple is an expression which consists of a subject node, predicate and an object node. *QLever* represents its knowledge base data also as graphs. Axel Lehmann describes in his thesis his approach to generate such RDF triples from OSM data to fill the knowledge base of *QLever* [5].

2.2 Data Rendering

The open source project *Datashader* is a graphics pipeline and is designed to work with large datasets [6]. It is written in Python but compiled to machine code to plot even millions of data points efficiently on a map. While this approach might be more suitable for data analysis, plotting map data on a canvas are a disadvantage for this project.

For *Leaflet* itself, there is a plugin for clustering called *Leaflet.markercluster* [7]. It is a powerful tool with lots of features for clustering. For smaller data sets (< 100.000) it is more than sufficient in terms of the user experience. Unfortunately, with larger data sets the initial loading times and the time for clustering becomes too long (see Chapter 4).

Therefore, we approached the problem with a filter function on the *QLever* side and with our own customized clustering algorithm on the web frontend.

3 Theoretical Analysis

In this chapter, we will give an overview of the functionalities of the web application, the backend and the used algorithms. The application depends heavily on the *QLever* engine. Therefore, we will first introduce the query language used in the engine (Section 3.1) and the backend itself (Section 3.2). We describe in Section 3.3 our implemented clustering algorithm and how it differs from the common k-means algorithm. In Section 3.4, we introduce the web frontend which consists of multiple parts: first we will describe the JavaScript library for mapping *Leaflet* in Section 3.4.1. Then we will introduce all used data structures (Section 3.4.2) and the map controls (Section 3.4.3). In Section 3.4.4, we show a complete overview of our web application.

3.1 SPARQL Query

SPARQL is a query language specifically designed to meet the use cases of RDF [3]. Listing 3.1 shows an example of a SPARQL query. This query describes in the `PREFIX` the first three namespaces `osmkey`, `geo` and `osm`. Then the `SELECT` part follows where it searches for an `osm_id` and `geometry` field. In the `WHERE` condition the RDF relations are specified that the `osm_id` should have an geometry field and the building type `university`. Lastly, it is being narrowed down with the `FILTER` function and the `envelope` key so that the objects should be contained by the rectangle defined with the `LINESTRING`.

Listing 3.1: SPARQL query: All university buildings in the bounding box of Freiburg

```
PREFIX osmkey: <https://www.openstreetmap.org/wiki/Key:>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX osm: <https://www.openstreetmap.org/>
SELECT ?osm_id ?geometry WHERE {
    ?osm_id osmkey:building "university" .
    ?osm_id geo:hasGeometry ?geometry .
    ?osm_id osm:envelope ?envelope .
    FILTER contained(?envelope, "LINESTRING(7.662006 47.903578, 7.930844
        48.071058)")
}
```

3.2 QLever Engine

QLever is a query engine for efficient combined search on a knowledge base [1]. The knowledge base is filled with RDF triples, for example with the subject *actor*, the predicate *movie type* and object *movie name*. To ensure efficient query operations the triples are sorted in all possible ways. The database of the engine is being filled with geometric data from OpenStreetMap. To access this data, the engine is using the query language SPARQL.

3.2.1 Well-known text

The well-known text (WKT) format was defined to represent a standard for spatial data by the Open Geospatial Consortium [8]. To present geometric data *QLever* uses only four strings:

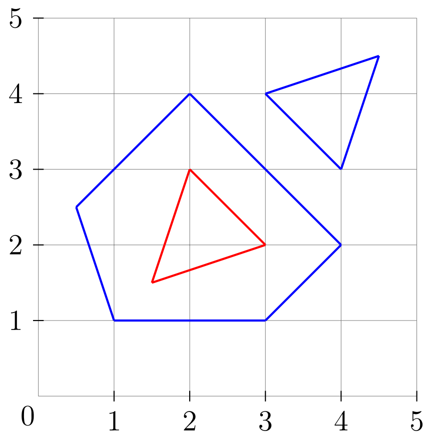

Figure 1: MultiPolygon example where the red triangle is being removed. WKT literal: `MULTIPOLYGON(((1 1, 3 1, 4 2, 2 4, 0.5 2.5), (1.5 1.5, 3 2, 2 3)), ((4 3, 4.5 4.5, 3 4)))`

1. The simplest spatial form is the point. The WKT representation of the point is `POINT(x0 y0)` with `x0` and `y0` as coordinates.

2. A path is just a series of points and will be represented as WKT literal `LINESTRING(x0 y0, x1 y1, ..., xn yn)`. The points in the path are separated by the comma.

3. A two-dimensional shape will be represented as a polygon. The WKT literal is `POLYGON((x0 y0, x1 y1, ..., xn yn), (a0 b0, ...))`. The first inner parenthesis with the `x` and `y` coordinates describe the area of this polygon. The second optional inner parenthesis with the `a` and `b` coordinates describe an area which will be removed from the first one.

4. The multipolygon is the most complex shape in this collection. The WKT literal `MULTIPOLYGON(((x0 y0, ..., xn yn), (a0 b0, ...)), ((...)))` represents such a shape. It can add multiple polygons together and remove parts respectively (Figure 1).

3.2.2 API GET Request

Listing 3.2 shows the corresponding response from an API GET request to the SPARQL query in Listing 3.1. The response consists of multiple parts from which the most important are:

1. ‘query’: The SPARQL query of this request.
2. ‘res’: An array of OSM data. In this example the URL to the relation of the OSM ID and a geometric shape described by coordinates as a WKT literal.
3. ‘resultsize’: The length of the res-array.
4. ‘selected’: The column names of the res-array.

Listing 3.2: JSON result of the API call: All university buildings in the bounding box of Freiburg

```
{
  "query": "...",
  "res": [
    [
      "<https://www.openstreetmap.org/relation/12450227>",
      "\"MULTIPOLYGON(((7.8472071 47.9936843, ...)),((7.8476325
          47.9943113, ...)))\"^^<http://www.opengis.net/ont/
         geosparql#wktLiteral>"
    ],
    ...
    [
      "<https://www.openstreetmap.org/way/101163690>",
      "\"LINESTRING(7.8445060 47.9936810, ...)\"^^<http://www.
         opengis.net/ont/geosparql#wktLiteral>"
    ],
    ...
  ],
  "resultsize": 298,
  "runtimeInformation": { ... },
  "selected": [
      "?osm_id",
      "?geometry"
  ],
  ...
}
```

3.3 Clustering

In this section, we will describe the clustering algorithm that is being used for the web application. Based on the *k-means* clustering described by J. MacQueen [9], we customized the algorithm for the specific task.

The algorithm has two phases. In the first phase, we describe how the initial centroids are being chosen (Section 3.3.1). In the second phase, we describe our implementation of the *k-means* clustering algorithm (Section 3.3.2).

3.3.1 Centroid Initialization

The first step is to choose the initial centroids before clustering. If the centroids are randomly chosen, it is possible that the clustering algorithm needs more iterations to converge.

Here, our focus is not data analysis but the overall user experience and loading times, so it is not important to reach the best possible solution. Therefore, we choose the initial centroids in linear time with the ‘Naïve Sharding’ method [10]. The sharding algorithm depends on a summation of attributes of all instances. Then these sums have to be sorted before the mean calculation of each attribute can take place. We customized and simplified this method for coordinates in Algorithm 1, to be as close as possible to the optimal solution but in a limited time.

We divide first the length of the input array by the number of clusters k to split the dataset into k equal-sized slices:

$$size = \left\lfloor \frac{|coordinateArray|}{k} \right\rfloor$$

Then for each slice of the coordinate array, we calculate the mean coordinate and add them to the initial centroid array.

3.3.2 K-Means Clustering

The basic idea of the k-means clustering algorithm is to pick a set of initial centroids and a number k for the amount of these centroids respectively clusters. Then to find a local optimum of the residual sum of squares, we need to alternate between the two steps [11]:

(A) Assign each element to its nearest centroid.

(B) Compute new centroids as average of the elements assigned to it.

We cannot predict what the size of every dataset will be. Choosing a k before knowing the size could lead to too many or too few clusters. Therefore, we decided to calculate a dynamic value for k in Algorithm 2 depending on the size a of given dataset:

$$k = \left\lfloor \log_2 \left(|coordinatearray|\right) \right\rfloor$$

For clarity purposes, the *coordinateArray* in this algorithm contains only the center of each object which are pre-computed. The objects themselves are attached to the center points but are not present in Algorithm 2 and in Algorithm 3.

Algorithm 1 Naive Sharding

```
function getCentroids($coordinateArray$, $k$)
    $step \leftarrow \lfloor length(coordinateArray)/k \rfloor$          ▷ Step length
    $centroids \leftarrow [\,]$                                          ▷ Initialize the centroids
    foreach $i \in \{0 \ldots k\}$ do                                    ▷ Make equal slices of the array
        $start \leftarrow step * i$
        $end \leftarrow step * (i + 1)$
        if $i + 1 = k$ then                                              ▷ End of the last slice
            $end \leftarrow length(coordinateArray)$
        end if
        $mean \leftarrow$ getMean($coordinateArray, start, end$)
        $centroids[i] \leftarrow mean$                                   ▷ Put the mean value into the
                                                                           centroid array
    end for
    return $centroids$
end function

function getMean($coordinateArray$, $start$, $end$)
    $step \leftarrow end - start$
    $mean \leftarrow [0, 0]$
    foreach $i \in \{start \ldots end\}$ do                              ▷ Iterate over the slice and cal-
                                                                           culate the mean coordinate
        $mean[0] \leftarrow mean[0] + coordinateArray[i][0]/step$
        $mean[1] \leftarrow mean[1] + coordinateArray[i][1]/step$
    end for
    return $mean$
end function
```

Furthermore, we define two global variables for maximum iterations *maxIteration* and maximum amount of objects per cluster *maxVal* before the cluster will be divided in subclusters. The *maxIteration* value is one of two termination conditions of the clustering algorithm. For data analysis this value would be higher but here it is intentionally low. This ensures a limited computation time for the clusters to balance the presentation time and quality. The other termination condition is the convergence of centroids (see Algorithm 2).

After the initialization of the centroids in Section 3.3.1, the center points are being assigned in Algorithm 3 to its nearest centroid. To calculate the distance between a object-center and a centroid we use the Euclidean Distance:

$$distance(x, y) = \sqrt{(x[0] - y[0])^2 + (x[1] + y[1])^2}$$

After the assignment of each center to its centroid, the clusters will be reviewed. In Algorithm 3 we compute for all center point per cluster a new centroid. If a centroid has no center points it will be discarded. This loop continues until one of the termination conditions is being met.

Afterwards, we review once again the final clusters. If one cluster contains more objects than *maxVal*, the clustering algorithm will run recursively on its objects to generate subclusters.

The reason for the addition of building subclusters is purely for the purpose of presentation of the objects. The user experience can get laggy when too many objects are being held in the background. With the objects assigned to clusters we can dynamically load and unload clusters which are not visible on the screen as described in the next Section 3.4.

Algorithm 2 K-Means Clustering

```
$maxIteration = 4$
$maxVal = 5000$
function kMeans($coordinateArray$)
    $k \leftarrow \lfloor log2(length(coordinateArray)) \rfloor$          ▷ Dynamic calculation of k
    $clusters \leftarrow [\,]$                                             ▷ Initialize the clusters
    $terminate \leftarrow$ false
    $iteration \leftarrow 1$
    $centroids \leftarrow$ getCentroids($coordinateArray, k$)
    while $terminate \neq$ true do
        $oldCentroids \leftarrow centroids$
        $centroidLabels =$ assignLabels($coordinateArray, centroids$)
        $centroids \leftarrow$ getNewCentroids($centroidLabels$)
        $iteration \leftarrow iteration + 1$
        $terminate \leftarrow$ terminator($oldCentroids, centroids, iteration$)
    end while
    foreach $group \in centroidLabels$ do
        if $lenth(group.objects) \geq maxVal$ then                         ▷ If too many objects..
            $subcluster \leftarrow$ kMeans($group.objects$)                ▷ ..cluster them again
            $group.subclusters \leftarrow subcluster$                      ▷ Add the subcluster to the
                                                                             cluster
        end if
        $clusters.push(group)$                                             ▷ Finally add the group to the
                                                                             main cluster array.
    end for
    return $clusters$
end function

function terminator($oldCentroids$, $centroids$, $iteration$)
    if $iteration \geq maxIteration$ then
        return true
    end if
    foreach $i \in \{0 \ldots length(centroids)\}$ do
        if $centroids[i] = oldCentroids[i]$ then
            return true
        end if
    end for
    return false
end function
```

Algorithm 3 Helper Methods: Assign Labels and Calculate new Centroids

```
function assignLabels($coordinateArray$, $centroids$)
    $labels \leftarrow \{\ \}$
    foreach $i \in \{0, \dots, length(centroids)\}$ do        ▷ Initialize the labels
        $labels[i] \leftarrow \{points : [\ ], centroid : centroids[i], subclusters : \{\ \}\}$
    end for
    foreach $i \in \{0, \dots, length(coordinateArray)\}$ do
        $centerPoint \leftarrow coordinateArray[i]$
        foreach $j \in \{0, \dots, length(centroids)\}$ do
            $currentCentroid \leftarrow centroids[i]$
            if $j = 0$ then
                $closest = currentCentroid$
                $closestIndex = j$
                $closestDistance =$ calculateDistance($centerPoint, closest$)
            else
                $newDistance =$ calculateDistance($centerPoint, closest$)
                if $newDistance < closestDistance$ then
                    $closest = currentCentroid$
                    $closestIndex = j$
                    $closestDistance = newDistance$
                end if
            end if
        end for
        $labels[closestIndex].points.push(centerPoint)$
    end for
    return $labels$
end function

function getNewCentroids($centroidLabels$)
    $newCentroidArray \leftarrow [\ ]$
    $newCentroid \leftarrow [0, 0]$
    foreach $group \in centroidLabels$ do
        $newCentroid \leftarrow$ getMeanPoints($group.points$)
        $newCentroidArray.push(newCentroid)$
    end for
    return $newCentroidArray$
end function

function getMeanPoints($points$)
    $mean \leftarrow [0, 0]$
    foreach $i \in \{0 \dots length(points)\}$ do
        $mean[0] \leftarrow mean[0] + points[i][0]/length(points)$
        $mean[1] \leftarrow mean[1] + points[i][1]/length(points)$
    end for
    return $mean$
end function
```

3.4 Web Application

In this section, we describe the functionality of the complete web frontend. First, we will give an introduction in the *Leaflet* library in Section 3.4.1. In Section 3.4.2, we show how the data from the *QLever* engine is being processed and the associated data structures. In Section 3.4.3, we describe the navigation on the map and how data is being loaded in the background. Lastly, we give a complete overview of the web application in Section 3.4.4.

3.4.1 Leaflet

There are a few libraries for mapping in JavaScript, for example *OpenLayers* or *Datamaps*, but because of the simple design, lightweight and usabilty we chose *Leaflet* as our foundation. *Leaflet* weighs just about 39 KB of JS (plus 4 KB in CSS) and has many features and plugins to use [2].

Listing 3.3 shows how easy it is to initialize a map. The map needs only a *DIV* container in the HTML file, a center and a zoom level. OpenStreetMap provides then the tilelayer with map tiles for each zoom level. Each zoom level z contains 4^z tiles starting with 0 where the whole world is represented by one 256x256 pixel tile. On zoom level 1 the world is divided by 4 256x256 pixel tiles and so on.

Adding shapes like markers or polygons and information popups to the map is just as easy as Listing 3.4 shows.

Listing 3.3: Map Initialization

```
// Initialize the map in the "map" div with a center and zoom
   level.
const map = L.map('map', {
  center: [48.00443, 7.83017],
  zoom: 8,
});

// Tilelayer of the map provided by OpenStreetMap.
const tilelayer = L.tileLayer(
  'https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
  attribution: '&copy;␣<a␣href="https://www.openstreetmap.org/
     copyright">OpenStreetMap</a>␣contributors',
}).addTo(map);
```

Listing 3.4: Adding objects to the map

```
// Creating a marker with coordinates and a popup.
const marker = L.marker([48.00443, 7.83017]);
marker.bindPopup("Hello! I am a marker");
marker.addTo(map);

// Creating a triangle polygon with coordinates and a popup.
const polygon = L.polygon([
  [47.993, 7.847], [47.983, 7.847], [47.993, 7.857]
]);
polygon.bindPopup("Hello! I am a polygon");
polygon.addTo(map);
```

3.4.2 Overview

As described in Section 3.2.1 *QLever* stores and returns the geometric shapes in the WKT format. *Leaflet* can only handle plain coordinates or a set as GeoJSON which is a format for geospatial data that supports the mentioned shapes [12]. Therefore we first need to translate the data from WKT format to GeoJSON with the help of the Terraformer project by the Esri Portland R&D Center [13]. The Terraformer.WKT package allows to transform the WKT literal into a GeoJSON objects with one line (Listing 3.5).

Then we can create a feature object from this GeoJSON object. This feature contains the type (point, linestring, polygon or multipolygon), the array with the coordinates of the shape and information content for the popup.

Listing 3.5: Converting WKT literals into GeoJSON objects

```
// Convert the WKT literal with Transformer.WKT package into a
   GeoJSON object.
const geoJSON = Terraformer.wktToGeoJSON(wktToParse);

// Create a feature with linked popup table and coordinates.
const feature = {
  'type': 'Feature',
  'properties': {
    'popupContent': createPopupTable(...),
  },
  'geometry': {
    'type': geoJSON.type,
    'coordinates': geoJSON.coordinates,
  },
};
```

3.4.3 Map Navigation

In Section 3.1, we indicated that *QLever* is able to load a subset of an area with a *contained* function. We use this function when a query result exceeds a certain limit. With this process we remember the initial zoom level for loading next areas so that the limitation is not being exceeded on dragging the map. We choose first one of the first objects in the result and zoom in on the map until the subset is below the limit. Figure 2 shows a schematic of a loaded area. The blue area is the visible area which is defined by the screen size of the browser. The red dashed area represents the loaded not visible area. The size of each of the eight areas is also defined by the screen size to ensure that the user does not have to wait for loading objects when he drags the map a little bit.

Loading new data happens in two cases:

1. The first case is when the visible map (blue area) is being dragged to the edge and one side hits the grey dotted area. The application generates then a new query for the gray area (Figure 3) depending on the already loaded area. In the course of this process the corner points are being stored to remember which area is already being loaded. Figure 4 shows as an example which area will be loaded in the next steps.

   It is important to note that a new area is only being loaded when the zoom level is not too low. The user interface will notify the user when the user zoomed out of the limit.

2. The second case is on the double-click event of the map. When the user zoomed out of the initial area it is possible to double-click on a coordinate. The application sets then again the initial zoom level and the previous loaded area is being discarded.

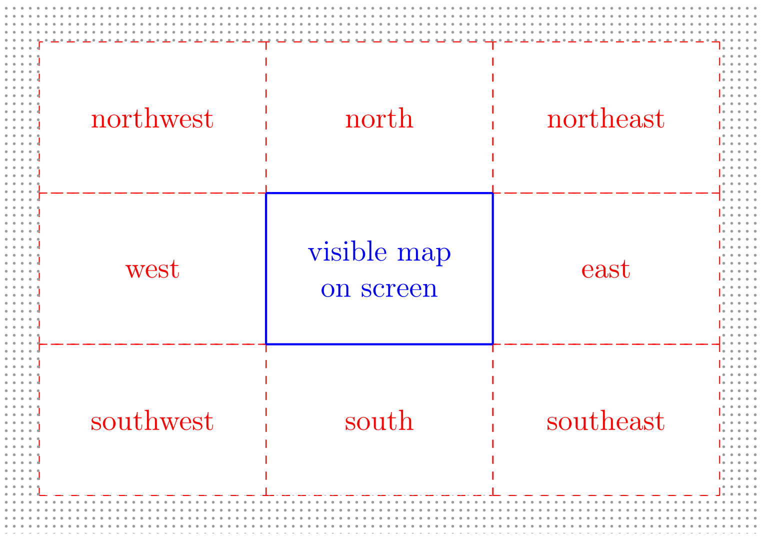

Figure 2: Schematic presentation of the loaded area of the map

3.4.4 Flowchart

Figure 5 shows the flow chart of the complete web application and how the algorithms and functions from the previous sections are implemented. First, the application takes two URL parameter `query` and `backend`. The `query` is described in Section 3.1 and the `backend` is the API URL to the *QLever* engine (Listing 3.6).

Listing 3.6: Example of the backend API URL

```
https://qlever.informatik.uni-freiburg.de/api/osm-planet
```

After encoding the `query`, we create a new query URL with a `LIMIT` clause. The function `loadTestQuery(url)` sends then a GET request to the *QLever* engine to get only the first few entries but with the result size of the whole query. The first object of the JSON result defines the initial center of the screen in `getFirstObject(result)`. In the next step, the application decides based on the result size whether it exceeds the maximum amount of objects to show (defined as `MAX_OBJECTS = 12.000`).

Case 1: If the current result size is larger than `MAX_OBJECTS` then the application

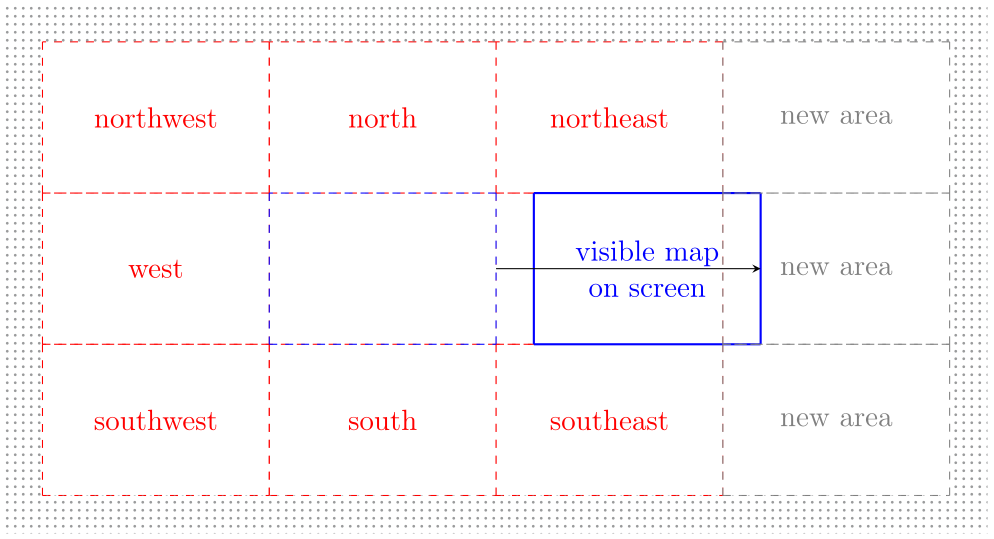

Figure 3: Schematic presentation of the loading new area of the map 1/2

zooms into the map by one level. `readEnvelope()` calculates then the new bounds (red area in Figure 2) for the new envelope query. This will be repeated until the result size of an envelope query is lower than `MAX_OBJECTS`. Then case 2 steps in.

Case 2: If the current (or total) result size is lower than `MAX_OBJECTS`, the last URL is being loaded without the `LIMIT` cause. In the `read(result)` function all objects are being converted from WKT literals into GeoJSON objects. These objects or geometric shapes are then being clustered by `kmeans(objects)`. Afterwards, each visible cluster is represented by a rectangle which contains the center points of its objects and a marker with the number of objects it holds.

When the objects are finally loaded, the map can then be navigated as described in Section 3.4.3. Additionally, the user can then click on a cluster rectangle to let the objects appear and right-click on it to disappear them again. Visible clusters disappear if the map is being dragged away from them and vice versa.

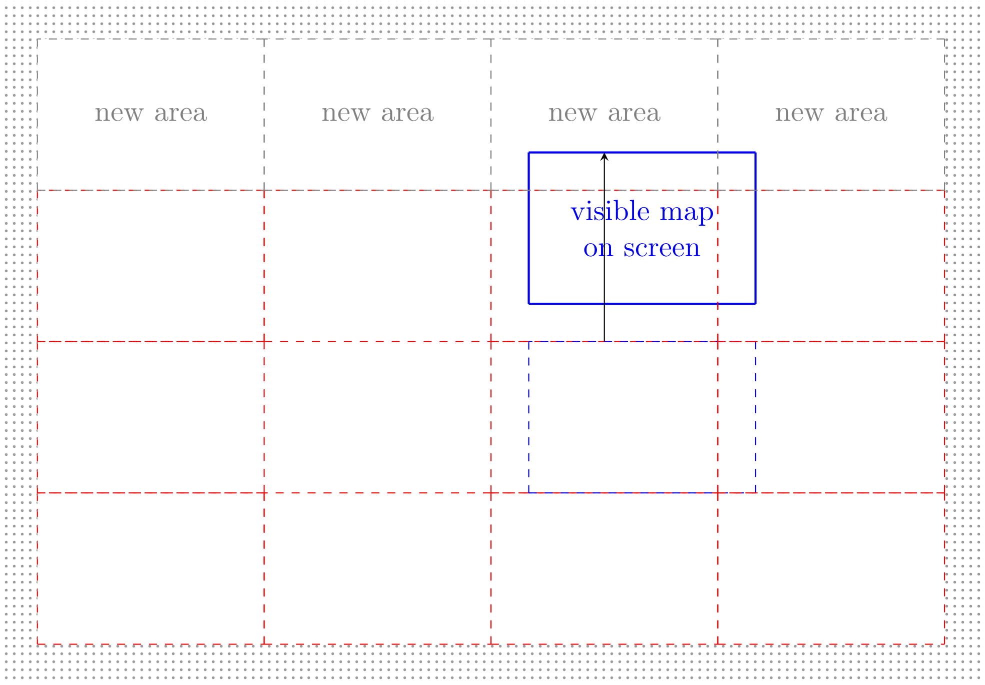

Figure 4: Schematic presentation of the loading new area of the map 2/2

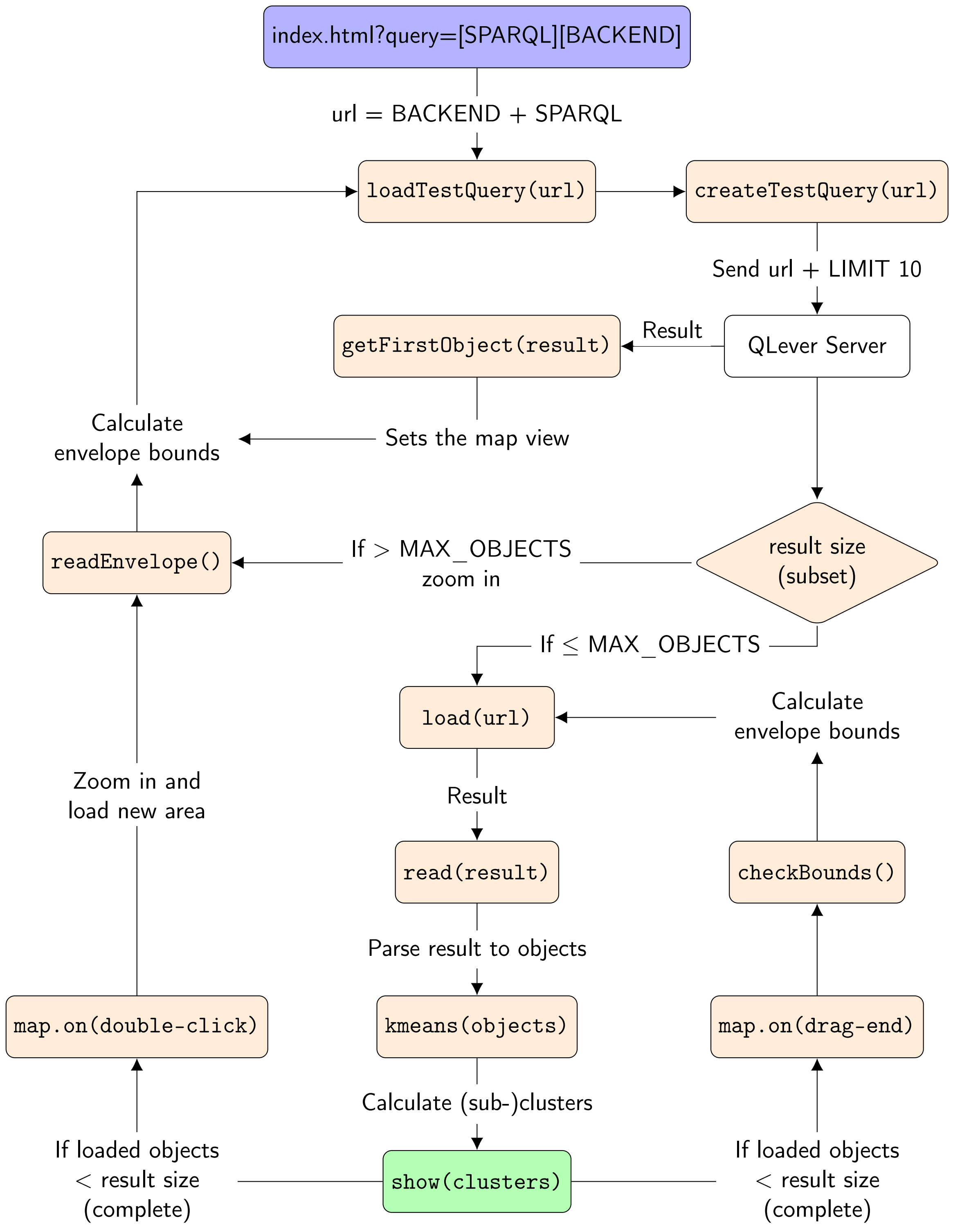

Figure 5: Flow Chart of the Web Application

4 Empirical Analysis

In this chapter we compare two aspects of the application. We compare the loading times between our implementation, a baseline solution and the plugin *Leaflet.MarkerCluster* (Section 4.1) and the overall user experience (Section 4.2).

All algorithms were written in JavaScript with the libraries *jQuery* and *Leaflet*. For the metrics we used the browser Chromium Version 92.0.4515.131 (64-bit) on a machine with Intel i5-8250U CPU @ 1.60GHz CPU and 24 GB RAM running on Pop!_OS 20.10 (based on Ubuntu). The average download speed of the used Internet connection was 57.49 Mbit/s.

4.1 Performance Test

For the performance test, we use the SPARQL query (Listing 4.1) which has a total result size of `57.992.788`. Although *QLever* is limiting the total results to a maximum of `100.000` objects, we can already see the growing response and render times in Table 1.

We modified the URL query with a `LIMIT` clause with `25.000`, `50.000` and `75.000` to see how response and render times would develop if *QLever* was sending more than `100.000` objects.

Our baseline solution is the most simple approach which is just loading all objects into the map. This shows already representative for the next approaches that the JSON result (memory usage) is growing more than linear compared to the number of objects. This is due to the fact that an object like a polygon consists of multiple points. The more complex the shape is the more space it takes. The solution with the *Leaflet.MarkerCluster* plugin takes even a longer render time than the baseline. With over `50.000` it takes already over 10 seconds for the initial load, which is not responsive. Our own clustering-only solution is indeed faster because it is less complex than *Leaflet.MarkerCluster*, but with over `75.000` objects in the JSON result the algorithm exceeds 9 seconds. Finally, our filtering combined with our clustering solution with the limitation of maximum `12.000` objects takes on average 946 ms (response + render time) for `5537` objects.

Listing 4.1: SPARQL query: All residential highways

```
PREFIX osm: <https://www.openstreetmap.org/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX osmkey: <https://www.openstreetmap.org/wiki/Key:>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?osm_id ?hasgeometry WHERE {
  ?osm_id osmkey:highway "residential" .
  ?osm_id rdf:type osm:way .
  ?osm_id geo:hasGeometry ?hasgeometry .
}
```

4.2 User Experience

Apart from the initial loading times, the overall user experience is most crucial for a web application. Despite that the baseline is faster in terms of rendering compared to

	response time (ms)	render time (ms)	memory	#objects
Baseline (simple loading)	1461.4	2132	9.5 MB	25000
	2920.0	4727.9	19.7 MB	50000
	4121.2	6426.6	28.4 MB	75000
	5401.8	8107.3	35.7 MB	100000
Leaflet.MarkerCluster	1432.7	2798.7	9.5 MB	25000
	2872.5	7654.0	19.7 MB	50000
	4134.2	14511.3	28.4 MB	75000
	5465.2	26200.3	35.7 MB	100000
Cluster only	1480.3	1816.5	9.5 MB	25000
(own implementation)	2884.1	3600.2	19.7 MB	50000
	4103.5	5170.1	28.4 MB	75000
	5417.5	6971.3	35.7 MB	100000
Filter + Cluster	196.8	206.1	302 kB	950
(own implementation)	289.1	293.0	861 kB	2786
	743.6	601.3	2.4 MB	7983
	692.4	414.7	2.4 MB	7983
	706.0	589.9	2.4 MB	7983

Table 1: Comparison of loading and render times

Leaflet.MarkerCluster, the navigation of the map starts to feel sluggish and laggy even with < `25.000` objects. The experience with *Leaflet.MarkerCluster* is the complete opposite. The plugin is very sophisticated and while the navigation is very good, the loading times are not. The cause is that it generates many clusters within clusters which results in longer render times the more objects are being loaded. Our final implementation delivers fast results in the shortest amount of time. The downside is that the user becomes attentive of the limited loaded objects when he zooms out.

5 Future Work

In this thesis, we have presented our implementation of efficiently rendering geospatial data on a map. Our main focus was to limit the maximum amount of objects being loaded at one time and cluster them.

The following points describe potential ways to enhance the usability and the response times:

1. The clustering function can be improved by adjusting the cluster shape. Instead of a simple rectangle it could get an approximate shape of all contained shapes.

2. The geospatial data in the *QLever* engine can be extended by adding and/or pre-computing more information:

   (i) The center point of the shape for the clustering algorithm.

   (ii) Which tile (for one of the higher zoom levels) contains this center point. This could result in better clusters without sacrificing computing time on the users end.

3. Pre-compute the clusters and send only the parent clusters to the user. Only after clicking on a subcluster the client sends a GET request for current visible screen.

The first solution is the easiest one, but it only delivers a better look and feel of the map. With solution 2, especially 2(ii), we could first send a query on the tile information and would get the complete geographical extent without limit.

Bibliography

[1] H. Bast and B. Buchhold, “Qlever: A query engine for efficient sparql+text search,” pp. 647–656, Proceedings of the 27th ACM International Conference on Information and Knowledge Management, 11 2017.

[2] V. Agafonkin, “Leaflet.” `https://github.com/Leaflet/Leaflet`, 2020. [Online; accessed 31-August-2021].

[3] S. Harris and A. Seaborne, “Sparql 1.1 query language.” `https://www.w3.org/TR/2013/REC-sparql11-query-20130321/`, 2013. [Online; accessed 31-August-2021].

[4] D. Beckett, “Rdf/xml syntax specification.” `https://www.w3.org/TR/REC-rdf-syntax/`, 2004. [Online; accessed 06-September-2021].

[5] A. Lehmann, “Creating a rdf knowledgebase from openstreetmap data,” Master’s thesis, Albert Ludwig University of Freiburg, 5 2021.

[6] I. Anaconda, “Datashader.” `https://datashader.org/`. [Online; accessed 06-September-2021].

[7] D. Leaver, “Leaflet.markercluster.” `https://github.com/Leaflet/Leaflet.markercluster`, 2018. [Online; accessed 31-August-2021].

[8] R. Lott, "Geographic information - well-known text representation of coordinate reference systems." `http://docs.opengeospatial.org/is/12-063r5/12-063r5.html`, 2015. [Online; accessed 31-August-2021].

[9] J. MacQueen, "Some methods for classification and analysis of multivariate observations," p. 281–297, Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, 1 1967.

[10] M. Mayo, "Toward increased k-means clustering efficiency with the naive sharding centroid initialization method." `https://www.kdnuggets.com/2017/03/naive-sharding-centroid-initialization-method.html`, 2017. [Online; accessed 31-August-2021].

[11] H. Bast, "Lecture notes in information retrieval," December 2018.

[12] H. Butler, M. Daly, A. Doyle, S. Gillies, T. Schaub, and T. Schaub, "The GeoJSON Format." RFC 7946, Aug. 2016.

[13] E. P. R. Center, "Terraformer." `https://terraformer-js.github.io/`. [Online; accessed 31-August-2021].