Bachelor's Thesis

Web Visualization Of Geographic Data Implementing A Thin Client Architecture

Hendrik Jan Henselmann

Examiner: Prof. Dr. Hannah Bast

University of Freiburg

Faculty of Engineering

Department of Computer Science

Chair for Algorithms and Datastructures

June 02nd, 2023

Writing Period

02. 03. 2023 – 02. 06. 2023

Examiner

Prof. Dr. Hannah Bast

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.

Waldshut Tiengen, 02.06.2023

Place, Date

Signature

Abstract

We present a system for efficiently visualizing large amounts of geographic data on a Web page. Our Web application implements a suitable architecture, the *thin client* architecture, that performs all computationally intensive operations on the server. These operations include requesting, processing and storing geographic data. The resulting image is then transmitted to the client and is finally displayed.

In this thesis, we specifically deal with the visualization of *OpenStreetMap* data. OpenStreetMap is an open-source project regarding geographic data. We utilize the query engine *QLever* to execute queries that return a subset of the data. We highlight this subset on an interactive world map.

Our results show that our approach is suitable for visualizing even millions of objects in seconds. It visualizes 2,000,000 points defined by 2,000,000 geographic coordinates in about half a second and 2,000,000 more complex geometries, like polygons, defined by about 8,500,000 geographic coordinates in about two seconds.

Zusammenfassung

Wir stellen ein System zur effizienten Visualisierung großer Mengen an geografischen Daten auf einer Webseite vor. Unsere Webanwendung implementiert eine geeignete Architektur, die "thin client" Architektur, die alle rechenintensiven Arbeitsschritte auf dem Server ausführt. Diese Arbeitsschritte beinhalten die Anfrage, Verarbeitung und Speicherung der geografischen Daten. Das dabei entstehende Bild wird dann an die Webseite übermittelt und muss dort nur noch angezeigt werden.

In dieser Arbeit beschäftigen wir uns speziell mit der Visualisierung von *OpenStreetMap* Daten. OpenStreetMap ist eine frei verfügbare Quelle von geografischen Daten. Zur Abfrage der Daten benutzen wir die Abfrage-Software *QLever*. Die Anfragen liefern eine Teilmenge des OpenStreetMap Datensatzes zurück, welche unsere Anwendung auf einer interaktiven Weltkarte hervorhebt.

Die Ergebnisse zeigen, dass unser Ansatz dafür geeignet ist mehrere Millionen Objekte in Sekunden zu visualisieren. Wir visualisieren 2.000.000 Punkte, die durch 2.000.000 geografische Koordinaten definiert sind, in ungefähr einer halben Sekunde und 2.000.000 komplexere Geometrien, wie zum Beispiel Polygone, die durch ungefähr 8.500.000 geografische Koordinaten definiert sind, in ungefähr zwei Sekunden.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Today we have access to an incredible amount of geographic data that is digitally available. However, the data alone is not sufficient. An appropriate system is needed to leverage the informative power of available data. Such a system is called *Geographic Information System (GIS)*. A GIS is responsible for storing, querying, analyzing and visualizing geographic data [1].

The use of a GIS usually requires the installation of an application. Furthermore, the inspected data, which is often huge, has to be locally available. Moreover, the computer running the GIS needs to fulfill the respective software requirements. Therefore, Web GIS applications gained popularity - applications that provide remote access to a GIS through the World Wide Web. They come with multiple advantages. But the main difference is that the demanding task of data processing is carried out by the computer providing the Web application. [1, 2]

This thesis proposes a Web application implementing one typical feature of a GIS, the visualization of geospatial data on an interactive map. Geospatial data is data with a spatial scope and geographic information about the location. We face the problem of a great number of objects that shall be visualized in a short amount of time. For that reason, an appropriate Web application architecture is required.

Section 1.1 introduces Web application terminology used throughout this thesis. Section 1.2 and Section 1.3 present the motivation and the objective of this thesis,

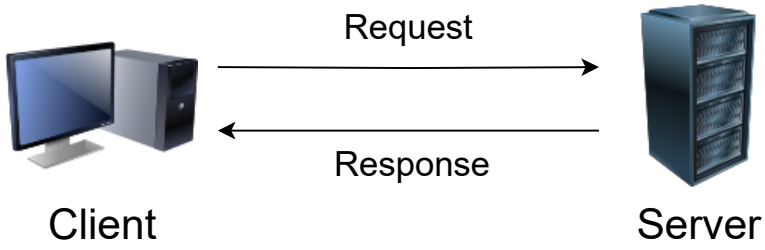

Figure 1: Client and Server. Illustrating two components of a Web Application.

respectively. Section 1.4 provides an overview of the approach elaborated in this thesis. Finally, Section 1.5 outlines the content of upcoming chapters.

1.1 Web Application Terminology

A *Web application* is an application that is accessible through the World Wide Web, usually by using a Web browser. Web applications rely on two fundamental components: the *server* and the *client*, illustrated in Figure 1. The server is a computer or a software program responsible for data storage and management. It provides Web pages and other resources in response to a client's request. The client is the device or software program requesting resources from the server, for example, the Web browser requesting a Web page. The part of a Web application that the user sees and interacts with is called the *frontend* of the application. Whereas the parts that are hidden from the user, mainly the server-side components, are commonly referred to as the *backend*. [1]

1.2 Motivation

OpenStreetMap[2] is one of the major open-source projects regarding geographic data. Numerous data querying tools are available for OpenStreetMap, including the query

[1] https://www.britannica.com/topic/Web-application
[2] https://www.openstreetmap.org

engine *QLever*. An essential feature improving the user experience of such a tool is the visualization of the query result. Especially interactive maps are a great enhancement, allowing the user to inspect the query result thoroughly. However, since the number of objects that must be visualized may exceed the millions, the naive approach of one-by-one object rendering within the user's Web browser is inadequate due to unacceptable long render times [3]. Even though a more sophisticated approach proposed by Denis Veil in [3] improves the render time, it is still restrained by Web browser capabilities and the resources of the client's computer.

1.3 Objective

We aim to develop a Web application implementing geospatial data visualization. The data results from the query engine QLever, executing queries about the OpenStreetMap data. Query results can contain geospatial information about all kinds of objects, for example, buildings, streets and even smaller objects like trees and post-boxes. The application's Web page shall display an interactive world map highlighting the geospatial information in the query result.

Figure 2 shows the visualization of an exemplary query result that contains trees and buildings. The images show the location of trees highlighted with green circles and the location and shape of buildings highlighted with blue polygons on top of a world map on two different zoom levels.

We want the map to be user-friendly, with fundamental map interactions, zoom and pan, and short display times. The display time refers to the time it takes to display updated visual information from when the user changes the view of the map.

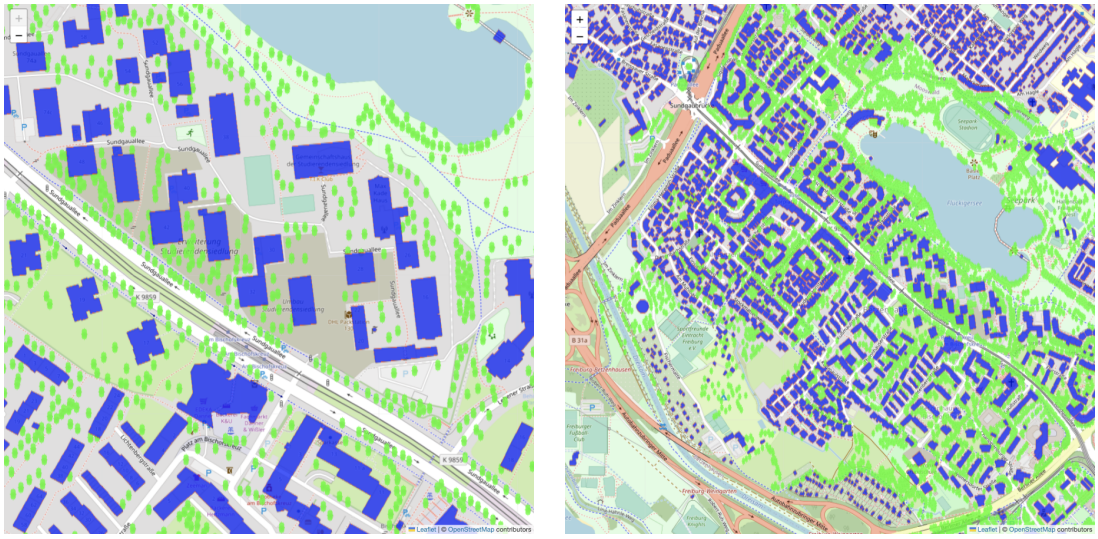

Figure 2: Highlighted buildings and trees. The images show the location of trees in green and the location and shape of buildings in blue. The interactive map in the background is provided by the mapping service Leaflet and OpenStreetMap.

1.4 Approach

This thesis proposes a Web application that visualizes query results containing geospatial data on top of an interactive world map. To cope with a potentially great amount of data, we use a Web application architecture called thin client, which performs the data management and data processing solely on the server. The client is just displaying the results received by the server. That approach avoids bottlenecks on the client side, such as the relatively slow execution speed of *JavaScript* code in the Web browser.

1.5 Chapter Overview

In Chapter 2, we describe and discuss related work. We present other available visualization tools for OpenStreetMap data and a paper concerned with Web GIS application architectures, shedding light on the software architecture of the application

proposed by this thesis. Chapter 3 introduces the background knowledge needed to understand the details of our work. Chapter 4 is the main body of this thesis. It presents our approach to how to visualize geospatial data on the Web. The performance of our approach is evaluated in Chapter 5. Finally, Chapter 6 summarizes this thesis and Chapter 7 lists potential improvements to our approach.

2 Related Work

In Section 2.1, we study a paper concerned with Web GIS application architectures in general. Two architectural designs are listed together with their advantages and disadvantages. The application proposed by this thesis implements one of these, the thin client architecture.

Many visualization tools are available for all the major geographic data resources. Section 2.2 presents common tools developed to visualize geospatial data on a Web page.

2.1 Web GIS: Technologies and Its Applications

In their paper "Web GIS: Technologies and Its Applications" [2], Alesheikh et al. study Web GIS technologies, especially their architectural design. They address multiple Web GIS architectures, including the *thin client* and the *thick client* architecture. These two architectures differ in the component that is processing data. That is illustrated in Figure 3. Applications implementing the thin client architecture perform the data processing on the server. The results are then sent to the client, which presents the results on the user interface. Thick clients, on the other hand, can process the data on their own, using the server mainly to request the data in the first place. These two approaches have strengths and weaknesses affecting the user experience. First, since the thin client's job is just the presentation of the server's computation results, the installation size of a thin client's frontend application or

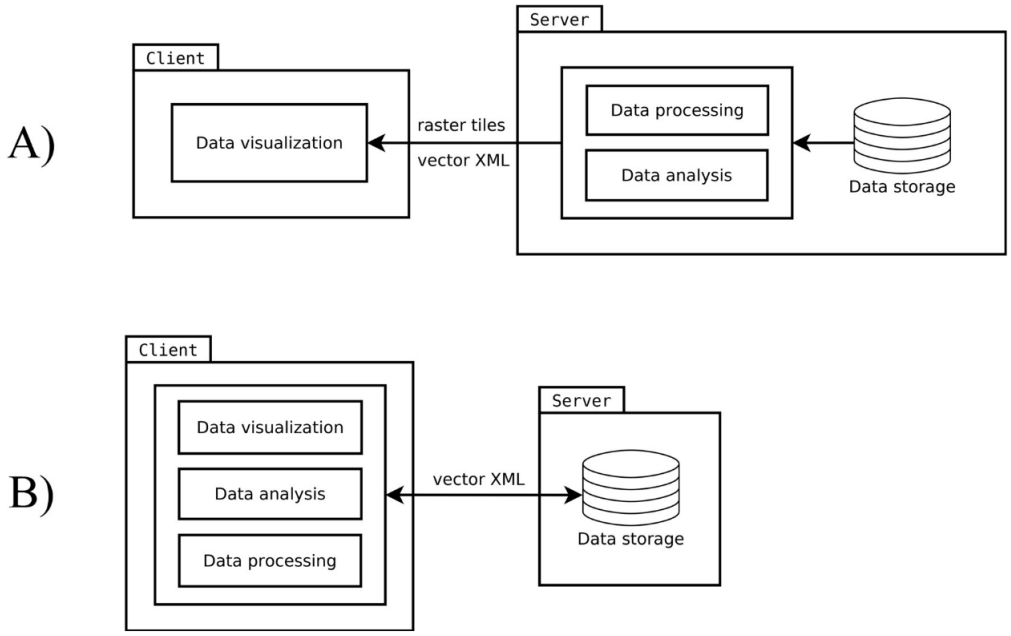

Figure 3: Web GIS Architectures. General overview of the thin client (A) and thick client (B) architectures. Source: [4]

Web browser extension is relatively small. Furthermore, the size of transmitted data and therefore, transmission times between client and server are usually lower on a thin client architecture. Computation results, for example, in the form of an image, are usually way smaller than the original data size. However, the frequency of data transmissions on a thin client architecture is higher because most user actions trigger a request for computations on the server. In a thick client architecture, the client carries out most computations. Requests are only needed if the user wants to work on a new data set. [2]

2.2 Web Visualization Tools for Geospatial Data

In this section, Web visualization tools are listed. Section 2.2.1 looks at the *Overpass API* and its Web application, *Overpass turbo*, providing an interactive map showing data from query results. Then, we introduce *OpenLayers* and *Leaflet* in Section 2.2.2.

These popular Web mapping services provide an interactive map with many valuable features. Nevertheless, their data highlighting features are very limited when the number of data points exceeds hundreds of thousands. Therefore, Denis Veil presents an approach utilizing a clustering technique that performs better than Leaflet [3]. We summarize his thesis in Section 2.2.3.

2.2.1 Overpass API and Overpass turbo

The *Overpass API*[1] is a tool designed to query OpenStreetMap data. The search criteria may include location, object type, tag properties and more. The response to a query request contains objects that match the search patterns. *Overpass turbo*[2] is a Web application using the Overpass API to answer queries. Additionally, the requested data is visualized on an interactive world map. Problems arise when the data set surpasses tens of thousands of data points. On such queries, scroll and zoom updates take multiple seconds, a major drawback to the user experience.

2.2.2 Leaflet and OpenLayers

Leaflet[3] and *OpenLayers*[4] are client-side mapping libraries for *JavaScript*. Both are open-source libraries developed by a large community. They provide many features regarding their primary focus, the interactive map. These features include data highlighting in various forms. The object of concern can be drawn by its geospatial information, which includes coordinates and shape. For example, a building can be represented as a polygon, illustrated in Figure 4. On the other hand, an object can also be highlighted using a marker that points to the object's location. That is illustrated in Figure 5. This method is sufficient to highlight the location of objects,

[1] https://wiki.openstreetmap.org/wiki/Overpass_API
[2] https://wiki.openstreetmap.org/wiki/Overpass_turbo
[3] https://leafletjs.com
[4] https://openlayers.org

Figure 4: Highlighting buildings using Leaflet. The image shows the location and shape of buildings in Freiburg, Germany, highlighted in blue. The interactive map in the background is provided by Leaflet and OpenStreetMap.

but it is inadequate to inform the user about the detailed distribution of objects, indicated by the image on the right.

On a greater scale, a marker cluster, provided by a Leaflet plugin[5], can give a nice overview of the local distribution of multiple objects. That is illustrated in Figure 6. A cluster is indicated by a circle containing the number of trees that the surrounding area of the circle contains. Zooming either splits clusters to show a more detailed distribution or merges clusters to show less detail but give a better overview of the distribution on a greater scale.

We measure the performance of Leaflet methods in Chapter 5. Visualizing hundreds of thousands of objects on an interactive map using Leaflet is unsuitable due to high rendering and response times that negatively affect the user experience.

[5]https://github.com/leaflet/Leaflet.markercluster

Figure 5: Leaflet markers indicating the location of trees. The images show the location of trees in Germany, highlighted with markers on three different zoom levels. The interactive map in the background is provided by Leaflet and OpenStreetMap.

2.2.3 Efficient Presentation of GeoSPARQL-Results

In the thesis "Efficient Presentation of GeoSPARQL-Results" [3], Denis Veil is concerned about visualizing subsets of the OpenStreetMap data in a Web browser, in particular larger subsets obtained from the query engine QLever that potentially exceed one hundred thousand objects. He concludes that the visualization tools provided by Leaflet are insufficient for that task. So he proposes a method that uses a clustering algorithm to manage the data stored in memory. Based on the area that is visible to the user, clusters are dynamically loaded and dropped. That is combined with an extension of the query utilizing a filter function provided by QLever to restrict the search area to the area of interest. This approach reduces the size of data in memory and optimizes data requests, decreasing the response and render time of the application. However, Denis Veil admits that the proposed algorithm limits the number of objects in memory to increase the application's performance, which is especially noticeable when the user increases the zoom level. [3]

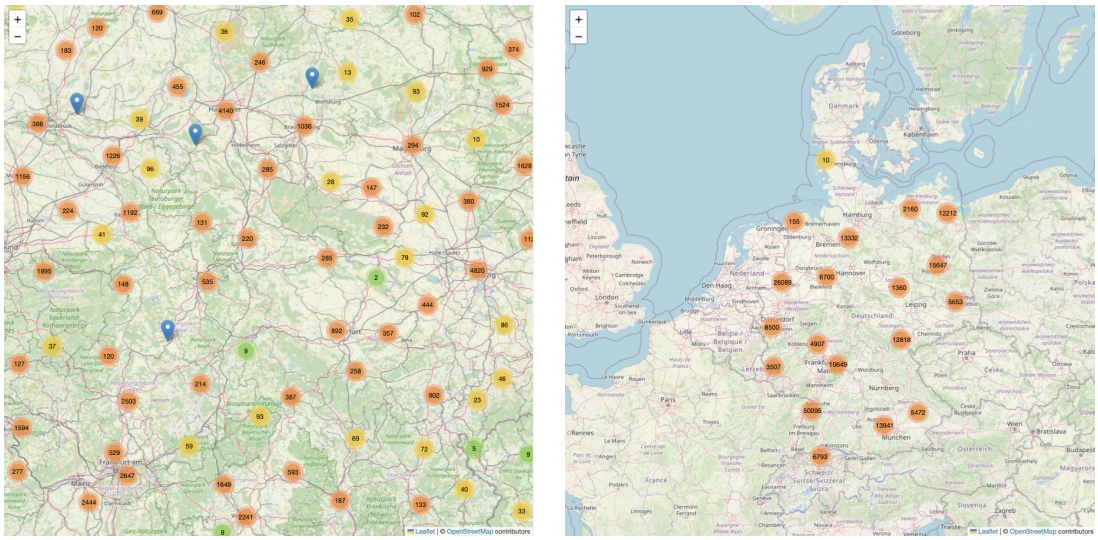

Figure 6: Leaflet marker cluster indicating the location of trees. The images show the location of trees in Germany displayed as a marker cluster on two different zoom levels. The interactive map in the background is provided by Leaflet and OpenStreetMap.

3 Background

This section provides relevant information necessary to understand our work, described in Chapter 4.

The web application we develop as a result of this thesis visualizes geospatial data. The visualized data results from queries about the *OpenStreetMap* database presented in Section 3.1. Section 3.2 presents the query engine *QLever*, which is used to execute those queries. The result of a query is a set of tuples that satisfy the formulated search patterns. Such tuples can contain geospatial information in *Well-Known Text* representation, illustrated in Section 3.3.

Geospatial data visualization on a flat map requires projecting the earth's spherical surface to a two-dimensional planar. That is the purpose of a *map projection* described in Section 3.4.

3.1 OpenStreetMap

OpenStreetMap[1] *(OSM)* is a project that aims to build a geographic database that is available for everyone for free. That is without having to pay for access or being bound to legal restrictions like copyright protection. These are the main differences from many other sources of geospatial data.

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

At first, the goal was to cover primarily street data. Nevertheless, over time the project expanded to cover many different types of objects, including buildings, post-boxes, and even individual trees. The data is collected mainly by volunteers. Additionally, governmental bodies and commercial organizations also contribute to the project. Many tools were developed to facilitate the contribution process and also the accessibility of the data, for example, *Osmium*[2]. Anyone can add data to the project and edit the existing data. The editing history is recorded to cope with mistakes or vandalism, allowing uncomplicated rollbacks.

The three fundamental building blocks of the OpenStreetMap data are nodes, ways and relations. *Nodes* represent one point on the earth's surface, denoted by latitude and longitude. A *way* is an ordered list of nodes. It can represent linear features, such as streets, but it can also represent polygonal areas, such as buildings, in case the first and last node match up. A *relation* adds information about the relationship of the relation members. The members are represented as an ordered list containing nodes, ways, relations or a mix of those. Furthermore, any OpenStreetMap element, that is, a node, a way or a relation, may have one or multiple tags. A *tag* is a key-value pair adding information to one element.

3.2 QLever

QLever [5] is an efficient query engine developed by Bast et al. at the University of Freiburg. It can execute queries about the OpenStreetMap data. However, the OSM data format is not compliant with the expected data input format of QLever. QLever requires that the data is in the form of *Resource Description Framework (RDF)* triples. An RDF triple consists of a subject, a predicate and an object. We use the tool *osm2rdf* [6] to convert OSM data to a database consisting of RDF triples.

[2]https://osmcode.org/libosmium/

Listing 3.1 illustrates a set of RDF triples. It describes the entities: *personA*, *personB* and *apple*. The four predicates *type*, *name*, *likes* and *knows* add information about the entities and their relations. For example, Maria is the name of *personA*. She knows Tom, *personB*, who likes apples.

```
personA type person .
personA name "Maria" .

personB type person .
personB name "Tom" .
personB likes apple .

apple type fruit .

personA knows personB .
personB knows personA .
```

Listing 3.1: Exemplary RDF database. It describes two persons: Maria and Tom. They know each other.

QLever is an implementation of the *SPARQL Protocol And RDF Query Language*[3] *(SPARQL)*, which is basically the standard query language for RDF databases. Listing 3.2 shows an exemplary SPARQL query that can be executed using QLever. This query returns all trees present in the queried data. The first two lines define prefixes with the keyword **PREFIX**. These are shortcut abbreviations for the following *Uniform Resource Identifier*, identifying the resource location. In general, the result of a SPARQL query is in tabular form, with rows and columns. The attributes listed in the **SELECT** clause, in this case ?osm_id and ?geometry, are part of the result. They specify the columns of the resulting table. Note that variables in SPARQL are indicated with a question mark as a prefix. The rows of the resulting table contain information about those objects that satisfy the patterns in the **WHERE** clause. A full stop separates multiple patterns. The patterns are structured like RDF triples, but they include variables. In this case, the first pattern matches all

[3] https://www.w3.org/TR/rdf-sparql-query

OpenStreetMap IDs ?osm_id for which the underlying data contains a triple ?osm_id **osmkey:**natural "tree". That returns IDs of objects that are tagged as being a tree. Additionally, the corresponding ?geometry is matched by the second pattern. So every row of the result contains the ID and the geometry of an object tagged as a tree in the underlying data.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX osmkey: <https://www.openstreetmap.org/wiki/Key:>
SELECT ?osm_id ?geometry WHERE {
  ?osm_id osmkey:natural "tree" .
  ?osm_id geo:hasGeometry ?geometry .
}
```

Listing 3.2: SPARQL query. Returns ID and geometry of all trees present in the queried data.

Additionally, Bast et al. propose the user interface *QLever UI* [7] to facilitate using the QLever engine. Advanced auto completion allows people unfamiliar with SPARQL syntax or the queried data's content and properties to construct SPARQL queries.

3.3 Well-Known Text Representation

Well-Known Text (WKT) is a representation standard of 2-dimensional (2D) geometric objects. A WKT literal is a human-readable string literal. Four types of objects are relevant to our work: point, linestring, polygon and multipolygon. A *point* represents a point in the underlying two-dimensional coordinate system. A *linestring* is an object that consists of straight lines connecting multiple sequential points. These points form the WKT representation of a linestring. A *polygon* defines a polygonal area potentially together with a hole. A *multipolygon* is a set of multiple polygons.

Figure 7 shows examples of shapes that can be represented by a WKT literal. Point A can be represented as POINT (1 1), following the scheme POINT (x_0 y_0) with the

x coordinate x_0, the y coordinate y_0, and a space between them. A linestring is a collection of points connected by lines in the order they are listed. So the shape B can be denoted as LINESTRING (1 2, 2 2, 4 1, 5 1), following the scheme LINESTRING (x_0 y_0, x_1 y_1, …, x_n y_n) and $n \in \mathbb{N}$. Note that a comma separates the points listed in the linestring while the x and y coordinates are separated by a space. The shapes C and D can be represented as polygon. The difference is that D has a hole in it. The WKT representation of C can be denoted as POLYGON ((1 3, 3 4, 2 5, 1 4.5, 1 3)). The general scheme of a polygon without a hole is POLYGON ((x_0 y_0, x_1 y_1, …, x_n y_n, x_0 y_0)), which is similar to the representation of a linestring. However, a POLYGON specifies the enclosed area by connecting the points. Notice that x_0 y_0 is the first and the last point listed. Shape D can be written as POLYGON ((3 6, 6 3, 9 6, 6 9, 3 6), (5 5, 7 5, 7 7, 5 7, 5 5)). The scheme of a polygon with a hole is POLYGON ((x_0^a y_0^a, x_1^a y_1^a, …, x_n^a y_n^a, x_0^a y_0^a), (x_0^b y_0^b, x_1^b y_1^b, …, x_n^b y_n^b, x_0^b y_0^b)). The first part, namely (x_0^a y_0^a, x_1^a y_1^a, …, x_n^a y_n^a, x_0^a y_0^a), specifies the outer contour of the polygon, while the second part represents the hole that is not part of the described area, (x_0^b y_0^b, x_1^b y_1^b, …, x_n^b y_n^b, x_0^b y_0^b). Finally, a multipolygon is a combination of multiple polygons. We can represent the polygons C and D together with a single WKT literal: MULTIPOLYGON (((1 3, 3 4, 2 5, 1 4.5, 1 3)), ((3 6, 6 3, 9 6, 6 9, 3 6), (5 5, 7 5, 7 7, 5 7, 5 5))). The representation of the two polygons follows the scheme of POLYGON and they are separated by a comma.

The WKT representations of the shapes in Figure 7 were denoted with numbers of their corresponding coordinate system. Our work is concerned with real-world objects, such as buildings and streets. The data is provided in the WKT representation. The coordinates of those objects refer to points on the earth's surface and are typically denoted by latitude and longitude. Figure 8 shows a building in Freiburg, Germany, highlighted in blue. It can be represented with the WKT literal POLYGON (7.8124817 48.0138416, 7.8127070 48.0140107, 7.8129456 48.0138685, 7.8127203 48.0136994, 7.8124817 48.0138416).

3.4 Map Projection

Our earth is a three-dimensional object, approximately an ellipsoid. A point on the earth's surface is usually denoted by two coordinates, latitude and longitude, illustrated in Figure 9. Latitude represents the distance of a point to the equatorial line. It ranges from -90 degrees to 90 degrees, depending on how far south or north a point lies, respectively. The longitude represents the distance of a point to the prime meridian, which passes through Greenwich, England. It ranges from -180 degrees to 180 degrees depending on how far west or east a point lies, respectively. To portray the earth or parts of it on a two-dimensional plane, for example, an image, a *map projection* is used to project world coordinates to coordinates on the plane. In the case of an image, the plane coordinates correspond to pixels. Many commonly used map projections are distinguished by the properties they preserve. Some projections preserve the shape of objects, others preserve direction or distance. The appropriate map projection can be selected based on the desired properties.

We use the default map projection of Leaflet's interactive map. It is known as *Web Mercator Projection*[5] and it preserves angles locally, meaning that relatively small shapes compared to the earth, for example, buildings, match their shape in reality. One downside of the projection is that it is not defined for points that lie further north than roughly 85.05 degrees or further south than roughly -85.05 degrees. Nevertheless, that is a minor drawback since it just excludes points very close to the poles. Equation (1) and Equation (2) define the Web Mercator Projection. These functions yield the corresponding x and y coordinates on an image of the world for a point with latitude *lat* and longitude *long* in radians. Note that the x and y coordinates correspond to a coordinate system with an inverted y-axis. The *width* and *height* parameters determine the scale of the image the points are projected to. A common value for width and height is $265 \cdot 2^{zoomLvl}$. The zoom level of the map

[4]https://www.techtarget.com/whatis/definition/latitude-and-longitude
[5]https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames

is represented with *zoomLvl*. A higher zoom level presents more details, requiring more pixels to display the same area. In Leaflet, zoom levels range from 0 to 18.

$$x = \lfloor width \cdot \frac{long + \pi}{2\pi} \rfloor \tag{1}$$

$$y = \lfloor height \cdot \frac{\pi - \log_e(\frac{\pi}{4} + \frac{lat}{2})}{2\pi} \rfloor \tag{2}$$

The equations above project a point to the corresponding point on an image that covers the whole surface of the earth. However, especially for interactive maps, generating an image of a smaller area is essential. Figure 1 describes the algorithm that returns the image coordinates of a point on an image that shows a specific rectangular area of the earth's surface. The coordinates of the northwest corner and the southeast corner define this area. At first, image coordinates of the corner points and the point of interest are calculated according to an image that would show the whole surface of the earth. Then, the image coordinates of the northwest corner are used to calculate the image's offset, that is, to what extent the projected coordinates of the point of interest are shifted. Since the *width* and *height* of Equation (1) and Equation (2) correspond to the width and height of an image of the whole world, the projected coordinates need to be scaled to fit the desired image size specified by *imgWidth* and *imgHeight*. The scaling factor is calculated as the ratio of the desired image size to the image size of the specified area on an image of the whole world. Finally, these values, offset and scale, transform the point's image coordinates of an image of the whole world to coordinates of an image of the specified area.

Algorithm 1 Map projection to specific area

Input

$northWestCorner$	North west corner of the requested area
$southEastCorner$	South east corner of the requested area
$point$	The point that will be projected
$imgWidth$	Desired width of output image
$imgHeight$	Desired height of output image

Output

$mappedPoint$	Image coordinates of $point$

Assumption $point$ lies in requested area

function MAP_POINT($northWestCorner$, $southEastCorner$, $point$, $imgWidth$, $imgHeight$)

Map the corners and $point$ to an image of the whole world
$mappedNorthWestCorner \leftarrow$ WEB_MERCATOR_PROJECTION(northWestCorner)
$mappedSouthEastCorner \leftarrow$ WEB_MERCATOR_PROJECTION(southEastCorner)
$mappedPoint \leftarrow$ WEB_MERCATOR_PROJECTION(point)

Compute offset according to requested area
$xOffset \leftarrow mappedNorthWestCorner.x$
$yOffset \leftarrow mappedNorthWestCorner.y$

Compute scale according to requested area size and desired image size
$areaWidth \leftarrow mappedSouthEastCorner.x - mappedNorthWestCorner.x$
$areaHeight \leftarrow mappedSouthEastCorner.y - mappedNorthWestCorner.y$
$xScale \leftarrow imgWidth/areaWidth$
$yScale \leftarrow imgHeight/areaHeight$

Transform the point's image coordinates
$mappedPoint.x \leftarrow xScale \cdot (mappedPoint.x - xOffset)$
$mappedPoint.y \leftarrow yScale \cdot (mappedPoint.y - yOffset)$

return $mappedPoint$
end function

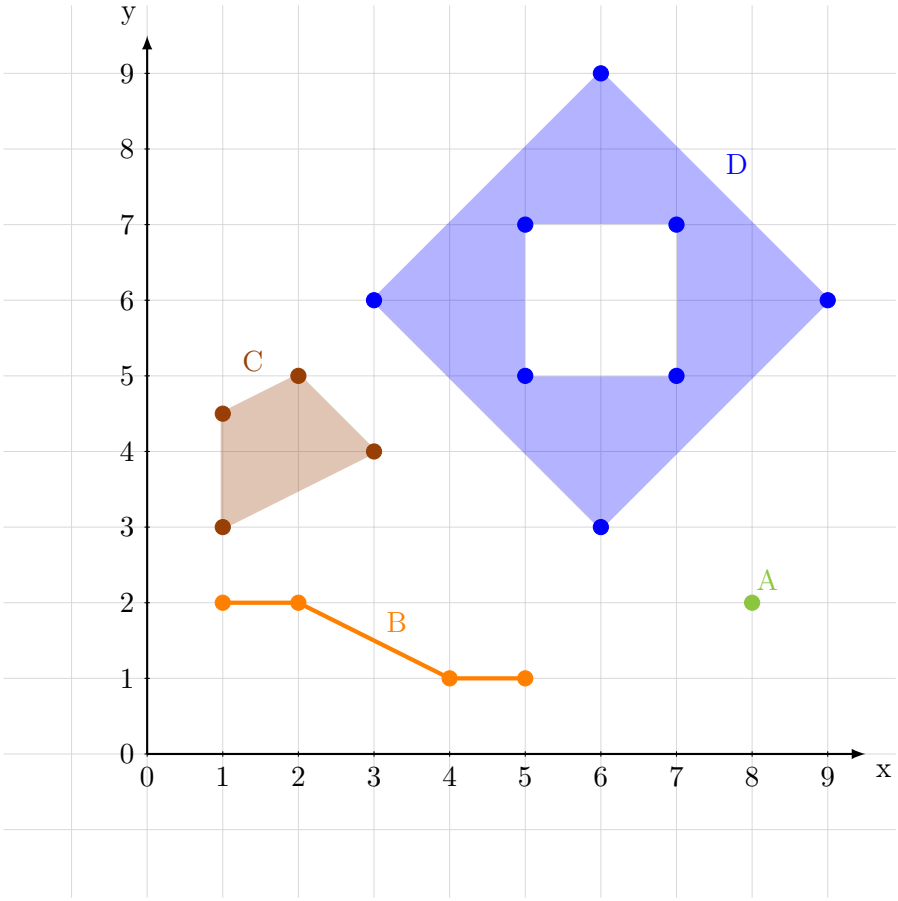

Figure 7: Shapes that can be represented as WKT literals.

A: POINT (1 1)

B: LINESTRING (1 2, 2 2, 4 1, 5 1)

C: POLYGON ((1 3, 3 4, 2 5, 1 4.5, 1 3))

D: POLYGON ((3 6, 6 3, 9 6, 6 9, 3 6), (5 5, 7 5, 7 7, 5 7, 5 5))

C and D as one shape:

```
MULTIPOLYGON (
    ((1 3, 3 4, 2 5, 1 4.5, 1 3)),
    ((3 6, 6 3, 9 6, 6 9, 3 6), (5 5, 7 5, 7 7, 5 7, 5 5)))
```

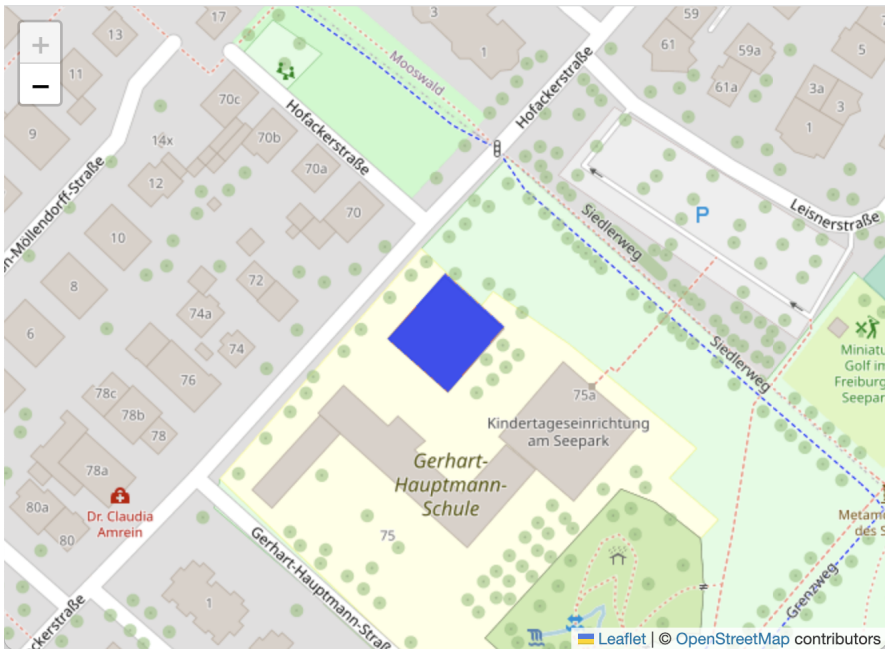

Figure 8: Building in Freiburg, Germany. The building is highlighted in blue. The background shows a map provided by Leaflet and OpenStreetMap.

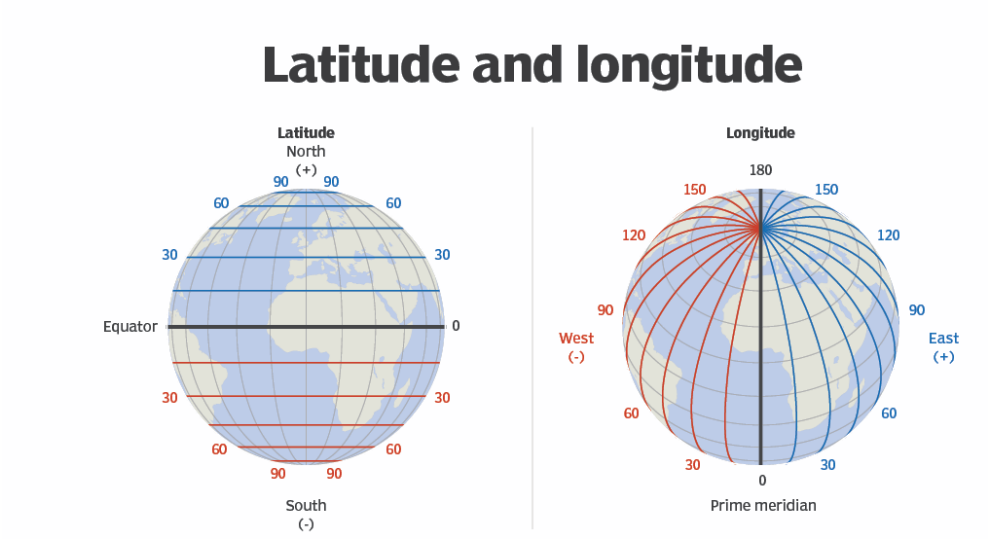

Figure 9: Latitude and Longitude. Source: TechTarget[4]

4 Approach

We develop a Web application implementing geospatial data visualization. The data is retrieved from the query engine QLever, executing queries about the OpenStreetMap data.

In this chapter, we present our approach in detail. In Section 4.1, we examine the concept of our approach. The proposed Web application consists of two components, the Web page and the server. Section 4.2 introduces the Web page of our application, representing the frontend. In Section 4.3, we present the implementation of the Server. We want our application to be as fast as possible to provide a pleasant user experience. Therefore, we are concerned with optimizations illustrated in Section 4.4.

4.1 Concept

Visualizing many million data points in the Web browser is a challenging task. *JavaScript* is the standard programming language of Web content. It is a dynamically typed, high-level language, which leads to slow code execution speed compared to other programming languages [8]. Additionally, the client's computer executes JavaScript code on a browser Web page. So the execution speed depends on the resources of the client's computer.

The core concept of this thesis is to outsource the hard work of processing potentially large amounts of data to the server that is providing the Web application. This

procedure has multiple advantages. Instead of using JavaScript for computationally demanding tasks, one can choose a lower-level programming language that outperforms JavaScript. Furthermore, the client's computer is no longer a limiting factor. The server carries out critical, time-consuming computations. In our case, this server is a powerful computer provided by the chair of Algorithms and Data Structures at the University of Freiburg. Instead of visualizing the data, the client sends a request to the server, specifying the area of the map that is visible to the user. The server generates an image of the data that the specified area contains and sends it back to the client. The client places this image on top of an interactive world map. This form of work distribution is summarized as thin client architecture (Section 2.1).

4.2 Web Page

The application's Web page is written in *JavaScript*, *HTML* and *CSS*. It shows an interactive world map provided by the mapping package Leaflet (Section 2.2.2). The Web page takes one parameter: the query, which will be executed by the QLever engine to be able to visualize the returned result. This parameter is handed over to the Web page as a *URL parameter*. An URL parameter is specified by a key-value pair. The key for this parameter is *query* and the value is the desired query. The query can be generated using the QLever UI (Section 3.2).

The world map provided by Leaflet is interactive. It supports the default user actions zoom and pan, common for interactive maps. Zooming increases or decreases the proportion of the world's surface visible to the user. The *zoom-in* action results in a greater level of detail and inversely, the *zoom-out* action gives a better overview of the data, showing fewer details. On the other hand, panning shifts the visible area vertically, horizontally or a mix of the two. The user can inspect a specific area on the desired level of detail using those two actions, zoom and pan, together.

A Leaflet map is composed using layers. The *tile layer* provides the basis for our

application. It mainly displays buildings and streets together with names, such as street names and city names, on different zoom levels. This information can be retrieved from different providers. We use a tile layer provided by OpenStreetMap. The second layer we use, in addition to the tile layer, is the *image layer*. We use the image layer to display an image of highlighted data on top of the map. The desired target location of the image can be specified in terms of geographic coordinates. The user interactions zoom and pan are also applied to the image. It always covers the same specified area of the world and is therefore shifted at pan actions and increased or decreased in size on zoom actions.

4.3 Server

The server providing the Web application is responsible for data management and data processing. It is written in the programming language *C++*. We use the *Gnu Compiler Collection GCC*[1] to compile the source code of our application. Furthermore, we use the tool *CMake*[2] to facilitate software development, specifically the build process.

Section 4.3.1 gives an overview of the server's workflow, consisting of QLever response parsing, described in Section 4.3.2, and the image visualization process, presented in Section 4.3.3. In Section 4.3.4, we assess the time complexity of the image generation process.

4.3.1 Server Workflow

The server workflow is illustrated in Figure 10. The server is listening for incoming requests issued by its frontend Web page. A request triggers the generation of an image of the requested geospatial data. However, the client does not explicitly send

[1] https://gcc.gnu.org
[2] https://cmake.org

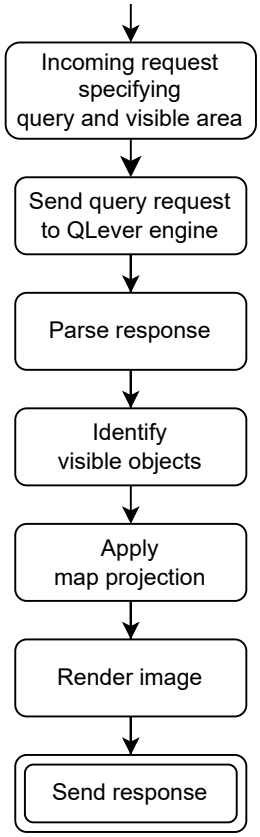

Figure 10: Server Workflow

that geospatial data to the server. The request specifies the query instead, which is executed by the QLever query engine backend. QLever responds with the query result containing geospatial information in the form of WKT literals (Section 3.3). But, this representation is a string literal that does not allow mathematical processing directly. Therefore, the objects received in WKT format are converted to an internal representation that defines the information as numerical data. Furthermore, the request of the client specifies the area that is visible to the user. Given that information, the server identifies the objects in this area. These objects' real-world coordinates, denoted by latitude and longitude, are then projected to the corresponding pixel coordinates on the resulting image using the Google Mercator Projection from Section 3.4.

4.3.2 Parsing a QLever Response

The default content type of a QLever response is text. This text contains the requested information about objects that match the patterns specified by the query. The purpose of our application is to visualize these objects using their geospatial information. Therefore, the query's SELECT clause must include a variable containing the geospatial information. This information is represented by a WKT literal (Section 3.3). The parser identifies a WKT literal based on the geometry type: point, linestring or multipolygon. A polygon is also declared as multipolygon. Based on the identified object type, it is reading one or multiple succeeding geographic coordinates and, in the case of a multipolygon, also the arrangement of coordinates, which is essential to be able to draw the object correctly. The coordinates are converted to numerical data and stored with information about the object type and the coordinate arrangement.

4.3.3 Image Generation

In this section, we describe the algorithm generating the requested image. It is illustrated in Figure 3. The algorithm takes two parameters: the geometries that are part of the query result and the area that is visible to the user. The algorithm consists of three steps. First, the visible objects are identified. Then, the geographic coordinates of these objects, denoted with latitude and longitude, are converted to pixel coordinates of the resulting image using the map projection presented in Section 3.4. Finally, the objects are drawn on an image with transparent background to be displayed on top of a world map.

The identification of the visible objects is illustrated in Figure 2. The algorithm takes two arguments: the geometries of all objects in the query result and the area visible to the user. It loops through the geometries and checks if the visible area entirely

Algorithm 2 Searching for visible objects

Input
geometries List of object geometries in query result
visibleArea Area visible to the user
Output
result List of geometries that are completely visible

function FIND_VISIBLE_OBJECTS(*geometries*, *visibleArea*)
result ← [] ▷ Empty list
foreach *geometry* ∈ *geometries* **do**
if *geometry* covered by *visibleArea* **then** ▷ Is the geometry visible?
result.push_back(*geometry*) ▷ Store geometry in output list
end if
end for
return *result*
end function

Algorithm 3 Generating the data highlighting image

Input
geometries List of object geometries in query result
visibleArea Area visible to the user
Output
img Image of visible objects

function GENERATE_IMAGE(*geometries*, *visibleArea*)

visibleObjects ← FIND_VISIBLE_OBJECTS(*geometries*, *visibleArea*)

projectedObjects ← [] ▷ Empty list
foreach *object* ∈ *visibleObjects* **do**
projectedObject ← MAP_PROJECTION(*object*, *visibleArea*)
projectedObjects.push_back(*projectedObject*)
end for

Initialize empty image *img*
foreach *projectedObject* ∈ *projectedObjects* **do**
img.draw(*projectedObject*)
end for

return *img*
end function

covers the geometry. If it does, the geometry is added to the returned list of visible objects. Otherwise, it is ignored.

Once all the visible objects are identified, an object's geographic coordinates corresponding to a location on the earth's surface are projected to a location on the resulting image. We use the Web Mercator Projection from Section 3.4 to perform this projection. The function *MAP_PROJECTION* in Figure 3 uses the algorithm presented in Figure 1 to map every geographic coordinate of an object to its corresponding image coordinate. Based on an object's geometry type, one or multiple coordinates must be projected.

Finally, the visible objects are rendered using their calculated image coordinates. We utilize the powerful computer vision tool *OpenCV*[3] for the drawing process. It provides functions to render different geometric shapes, like circles, lines and polygons. The background of the drawn image can be transparent, as well as the drawn shapes. That is necessary because we want to display the image of the highlighted data on top of a world map. Using a non-transparent background would hide the underlying map. Figure 11 shows two images of the same area of a map provided by Leaflet. The image on the right is additionally highlighting data.

On low zoom levels showing large proportions of the earth's surface, drawing objects according to their detailed shapes is pointless because they appear too small or do not even span multiple pixels of the generated image. Therefore, we introduce a second data highlighting mode activated on low zoom levels. On high zoom levels, objects' shapes are still rendered as before. However, on low zoom levels, a heat map is created instead. Figure 12 shows what such a heat map looks like. The highlighted data contains all buildings in Freiburg, Germany, present in the OpenStreetMap data. The images differ in zoom level. The ones in the first two rows are rendered as heat maps. That gives an excellent overview of the distribution of objects in the visible area. Regions highlighted in red indicate a higher density of objects, while the density

[3]https://opencv.org

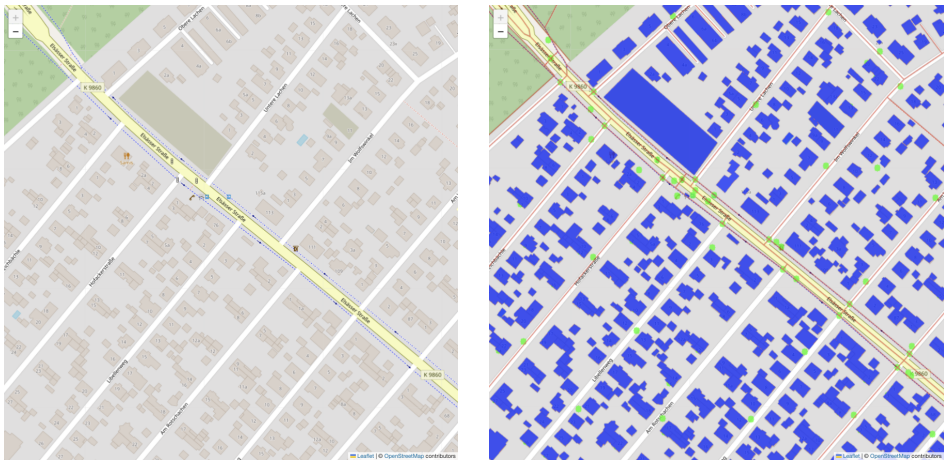

Figure 11: Leaflet map with and without highlighted data. Both images show the same area of an interactive map provided by Leaflet and OpenStreetMap. The image on the right is additionally highlighting data. Points are highlighted as green circles. Lines are drawn in red and polygons in blue.

in blue regions is relatively low. The images in the third row show that the buildings are still drawn in detail on high zoom levels.

Figure 13 shows a flowchart of an adapted server workflow. Black elements show the original workflow of Figure 10 and elements regarding the heat map creation are colored in red. If an image with a zoom level lower than the specified threshold is requested, the number of objects per image pixel is counted after the map projection is applied. Then we use the library *libheatmap*[4] to generate a heat map image visualizing the data.

4.3.4 Time Complexity

In this subsection, we analyze the theoretical time complexity, also known as run time, of the image-generating algorithm in Figure 3, precisely the asymptotic run time in the worst case. First, we assess the run time of the search for visible objects

[4]https://github.com/lucasb-eyer/libheatmap

illustrated in Figure 2. The algorithm contains one for-loop. It iterates over all geometries in the query result. So the run time depends on the number of geometries in the query result. We call this number n. Every iteration, the algorithm checks whether the geometry is visible and if it is, it is added to the resulting list of all visible objects. To check if a geometry is completely visible, the algorithm must check for every geographic coordinate in the definition of the geometry whether it is visible. So the run time depends on the number of coordinates in a geometry's definition. This number can be arbitrarily large, but it is finite. That is also true for n, the number of geometries in the query result. We define m as the maximum number of geographic coordinates of all geometries in the query result. Therefore, m is the upper bound of the number of coordinates in a geometry. The visible area is a rectangle defined by the northwest and southeast corners. The relative position of the coordinate to these corners determines the visibility of a geographic coordinate. Calculating a coordinate's relative position is not dependent on any variable. Therefore, it is a constant time operation. Adding a visible geometry to the *result* list takes amortized constant time, assuming the list is implemented as a dynamic array. In conclusion, the run time of one iteration of the for-loop only depends on the number of geographic coordinates of the geometry, which is limited by m. The loop body is executed n times, resulting in a run time of $\mathcal{O}(n \cdot m)$ of the algorithm searching for visible objects.

The mapping of the visible objects to their corresponding image coordinates contains a for-loop iterating over all visible objects. In the worst case, all objects are visible. Therefore, the upper bound of the number of loop iterations is n. The *MAP_PROJECTION* function applies the *MAP_POINT* function in Figure 1 to every geographic coordinate defining the object's geometry. The upper bound of the number of coordinates defining a geometry is m. The time complexity of the *MAP_POINT* function is constant since it is composed of constant time operations only. Assuming that the list of projected objects, *projectedObjects*, is implemented as a dynamic array, adding an object takes amortized constant time. So the whole

process of applying the map projection to all the visible objects has a time complexity of $\mathcal{O}(n \cdot m)$.

Assessing the time complexity of *OpenCV*'s drawing functions goes beyond the scope of this thesis. We can conclude that identifying and projecting visible objects of a query result has an asymptotic time complexity of $\mathcal{O}(n \cdot m + n \cdot m) \sim \mathcal{O}(n \cdot m)$.

4.4 Optimizations

In order to improve the user experience of our application, we introduce optimizations that speed up various sub-processes. We reuse computation results. That process, described in Section 4.4.1, is known as caching. In Section 4.4.2, we introduce the concept of code parallelization. We extend the requested area to prevent the server from getting overwhelmed with requests triggered by frequent but small panning actions. This adjustment is presented in Section 4.4.3.

4.4.1 Caching

Caching is the process of storing computation results once a computation is performed. A cached result can be reused whenever the same computation would otherwise be necessary again. That decreases run time, especially for time-consuming computations.

Query Result Caching Query results are cached so that a sequence of requests about the same query does not always trigger a query request to the QLever engine, but only once, the first time the user requests a new query. That is especially useful for the interactive map setting because every user action, that is, zoom or pan triggers a request. Therefore, caching the query result allows for shorter response times. Figure 13 shows the optimized server workflow. The elements in blue correspond to

the query result caching. Compared to the unoptimized version, colored in black, the two steps, query answering by the QLever engine and response parsing, are only executed once the user issues a new query. However, the query result is additionally stored for follow-up requests about the same query.

QLever Data Caching QLever provides the option to answer a query request with *QLever IDs* corresponding to the internal QLever data. These QLever IDs are of uniform size and can be transmitted in binary form, speeding up the data transmission process. To construct a query result out of the QLever IDs, it is necessary to build and store the mapping of QLever IDs to internal QLever data. This process can take multiple hours, depending on the size of the QLever data. Nevertheless, once the mapping is constructed for the first time, it is serialized on the server. That has the advantage that upon a restart of the application, the data can be read from a file, which is generally faster than the data transmission between applications.

4.4.2 Code Parallelization

We parallelize data manipulation tasks using the library *OpenMP*[5], utilizing the full power of the multi-core CPU architecture of the server running the application. Specifically, we are parallelizing for-loops working on the C++ data structure *vector*. The downside of task parallelization is that thread safety becomes a concern. Two code instances must not manipulate shared resources at the same time because that causes nondeterminism of the outcome, depending on the unpredictable order in which the manipulations happen. To prevent this nondeterminism, we either ensure that different code instances always work on distinct parts of a data structure or never work on the same data structure at the same time.

[5]https://www.openmp.org

4.4.3 Extension of Requested Area

The client requests an image of the visible area. By extending the size of this area, we avoid frequent consecutive requests for insignificantly small panning actions. That increases the expected time until a new image is requested because it reduces the server's workload.

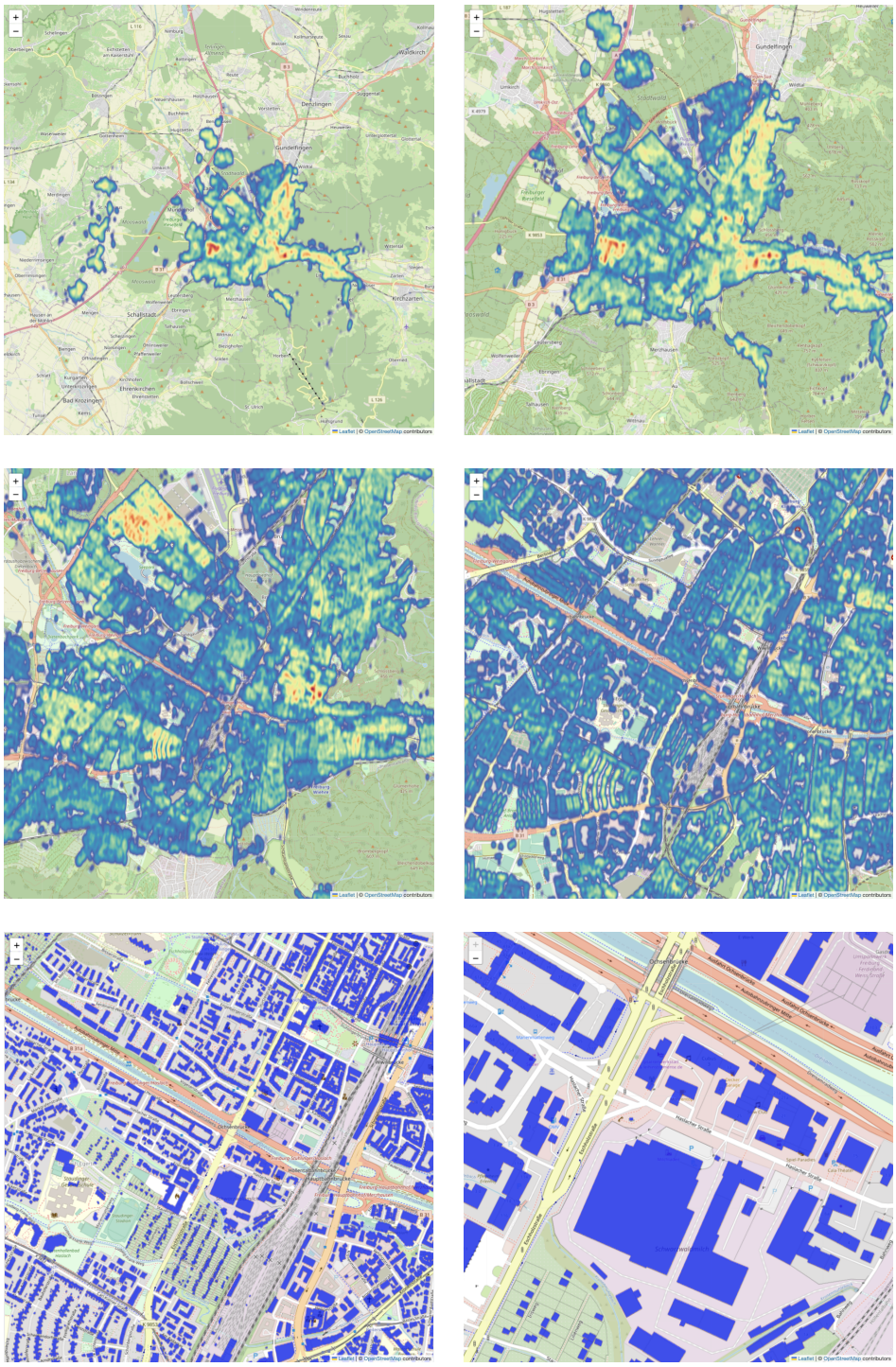

Figure 12: Images of highlighted data depending on zoom level. The images show the highlighted data as heat map on lower zoom levels and detailed shapes on higher zoom levels. The data is about buildings in Freiburg, Germany. The map in the background is provided by Leaflet and OpenStreetMap.

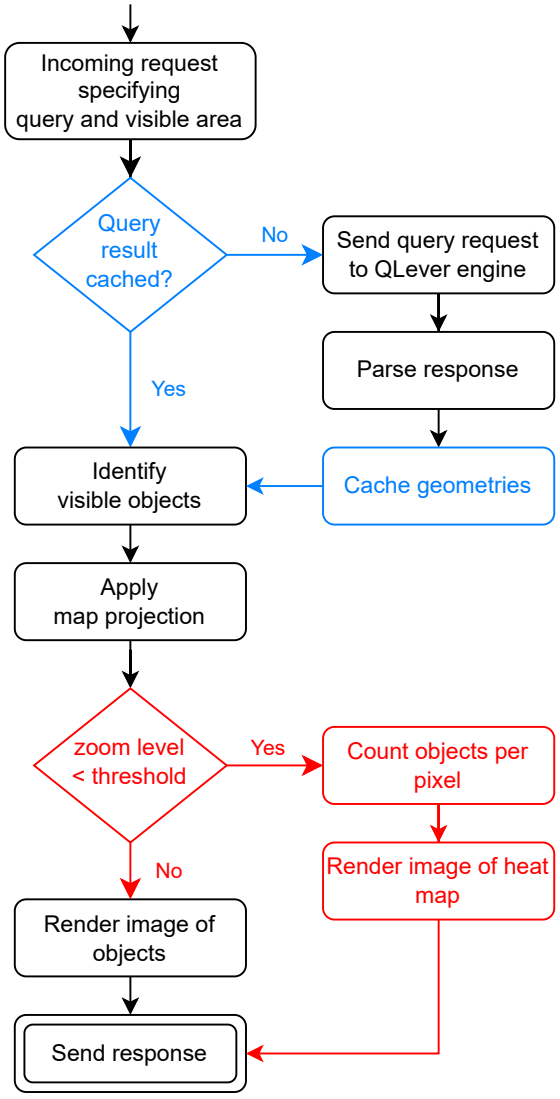

Figure 13: Adapted Server Workflow. Elements colored in black correspond to the original server workflow in Figure 10. Elements colored in red represent adjustments to visualize a heat map on low zoom levels. Blue elements introduce the optimization of QLever result caching.

5 Evaluation

In this chapter, we evaluate our approach empirically. The hardware, software and data used to conduct measurements are listed in Section 5.1. In Section 5.2, we describe the measured values. Section 5.3 presents the two queries that we use for the empirical evaluation. Section 5.4 provides the evaluation baseline. Finally, the results are reported and analyzed in Section 5.5.

5.1 Setup

For our measurements, we host the Web application on a computer of the Chair of Algorithms and Data Structures at the University of Freiburg. This computer has an *AMD FX8150* 8x3.6 GHZ CPU, 32 GB of *DDR3* RAM, a 128 GB SSD and a 3 TB HDD. The operating system is *Ubuntu 22.04*. It is the server of our Web application. The Web application is accessed through the World Wide Web using another computer, acting as the client. That computer has a 2,7 GHz *Quad-Core Intel Core i7* CPU, 16 GB of *DDR3* RAM, an *Intel Iris Plus Graphics 655* 1536 MB GPU and 500 GB of hard disk space. The installed operating system is *Mac OS 13.3.1*. We use the Web browser *Google Chrome*, version *113.0.5672.126*. We are using *Docker*[1] for the development and execution of the code, ensuring portability and reliability.

[1] https://www.docker.com

We are hosting an instance of the query engine *QLever* on the server. Furthermore, we download and process an OpenStreetMap data set using the tool *osm2rdf* (Section 3.2), converting the OSM data to RDF triples (Section 3.2). The data specifically covers geospatial information about Germany, allowing QLever to execute queries about the country.

5.2 Measured Values

Our proposed Web application issues a request every time the user adjusts the zoom level, zooming in or out and when the user is shifting the visual area to an area not covered by the previous image. We simulate the data inspection process of a regular user by zooming and panning the map.

We are concerned with a user-friendly experience using our application. Long *display times* are a major concern regarding interactive maps. Display time refers to the time it takes to display the requested information on the Web page after the user performs an action. The display time also includes the *data transmission time*. The data transmission time is the time needed to send the image request to the server and the time it takes to transfer the generated image to the client. The data transmission time depends on the internet connection speed and the distance between the client and the server. Therefore, we also measure the *generation time*. That is the time it takes for the server to generate the image, which is independent of the client-server connection.

We differentiate between *initial* and *regular* time. The initial time corresponds to the initial visualization request of a query. The regular time corresponds to follow-up requests regarding the same query. The initial time regarding our application includes query execution and result caching; regular time refers to the time of zoom and pan actions using the cached query result.

We conduct each measurement several times, returning a series of values $x_0, x_1, \ldots, x_N$ with $N \in \mathbb{N}$. We calculate the *average* $\hat{x}$ (Equation (3)) and the *corrected sample standard deviation* σ (Equation (4)) of the measured values [9] and report the results in the form $\hat{x} \pm \sigma$. The corrected sample standard deviation is an estimator of the *standard deviation* of the measured values. The standard deviation is a measure of the dispersion of the measured values around the average value $\hat{x}$. Therefore, it is an indicator of the uncertainty of measurements.

$$\hat{x} = \frac{1}{N} \sum_{i=0}^{N} x_i \tag{3}$$

$$\sigma = \sqrt{\frac{1}{N-1} \sum_{i=0}^{N} (x_i - \hat{x})^2} \tag{4}$$

It is important to mention that the initial generation and display time depends on the QLever engine executing the query. QLever is also caching query results internally, affecting the initial response time of our application. Furthermore, the reported times depend on the number of objects and object-defining coordinates visible to the user. The image generation time grows with the number of objects that must be rendered. Zooming on an area that contains very few objects reduces the image generation time. Therefore, we pay attention to always cover a large proportion of the data.

5.3 Queries

First, we use the query from Listing 3.2 to evaluate our application. The query returns all the trees in the queried data. Trees have the desirable property that their geometry is a point defined by a pair of latitude and longitude. We can limit the number of objects returned from a QLever query but not the number of object-defining

coordinates. Requesting objects represented as points enables us to measure loading times for specific query result sizes because a point is always represented by one coordinate consisting of latitude and longitude.

Points are just one of the three shapes our application has to deal with. To assess the performance of our application in case objects of all possible shapes are requested, we also measure the times for the query in Listing 5.1. It returns all objects in Berlin, Germany, which provide geospatial information. Since the representation of a linestring or a multipolygon can be arbitrarily large, the number of objects in the query result is inappropriate to determine the size of the visualized data. Therefore, we also report the number of geographic coordinates included in the objects' representation.

```
PREFIX ogc: <http://www.opengis.net/rdf#>
PREFIX osmrel: <https://www.openstreetmap.org/relation/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
SELECT ?osm_id ?geometry WHERE {
  osmrel:62422 ogc:contains ?osm_id .
  ?osm_id geo:hasGeometry ?geometry
}
```

Listing 5.1: SPARQL query. Returns ID and geometry of all objects located in Berlin that provide geospatial information.

5.4 Baseline

In this section, we report the display times of Leaflet for displaying trees. This is a client-side approach implementing the thick client architecture (Section 2.1). The data is requested, stored and processed by the client. We report times of two different methods: displaying markers and a marker cluster (Section 2.2.2). The corresponding query returning the trees is illustrated in Listing 3.2. The recorded times represent a

baseline for the Web application proposed by this thesis.

Number of objects	Regular display time (in ms)
1,000	420 ± 173
2,000	678 ± 149
4,000	1,134 ± 66
6,000	1,770 ± 64
8,000	2,418 ± 84

Table 1: Display time for points using Leaflet markers. Display time refers to the time it takes to process and display the geospatial information which is requested by the user. The average $\hat{x}$ and corrected sample standard deviation σ of measured values are reported in the form $\hat{x} \pm \sigma$. The visualized objects are trees, which are represented as points. Listing 3.2 shows the corresponding query.

Table 2 shows regular display times in case every single tree is displayed as a marker. The regular display time exceeds one second for a result size of 4,000 trees. It is almost two and a half seconds for 8,000 trees. That means the application needs more than two seconds to display the map with updated information when the user issues a zoom action, for example.

Number of objects	Initial display time (in ms)	Regular display time (in ms)
10,000	319 ± 38	268 ± 96
50,000	2,518 ± 95	271 ± 112
100,000	8,185 ± 469	294 ± 109
200,000	38,342 ± 1,355	457 ± 148

Table 2: Display time for points using Leaflet marker cluster. Display time refers to the time it takes to process and display the geospatial information which is requested by the user. The initial display time, as opposed to the regular display time, also includes the request of the data in the first place. The average $\hat{x}$ and corrected sample standard deviation σ of measured values are reported in the form $\hat{x} \pm \sigma$. The visualized objects are trees, which are represented as points. Listing 3.2 shows the corresponding query.

Table 1 shows initial and regular display times in case trees are displayed using a marker cluster. The regular display time for 200,000 trees is less than half a second. That is less than the regular display time of displaying 2,000 trees using markers without a cluster. The reported regular display times indicate that the marker cluster approach outperforms the approach of displaying every single tree with a marker. The downside of the marker cluster approach is that it has relatively high initial display times. That is the time it takes to display the information for the first time from when it is requested by the user. It takes more than eight seconds to initially display 100,000 trees and about 40 seconds to display 200,000 trees.

5.5 Results

In this section, we report measured generation and display times of our proposed Web application. The reported times correspond to the query results of the two queries presented in Section 5.3.

	Initial		**Regular**	
Number of objects	generation time (in ms)	display time (in ms)	generation time (in ms)	display time (in ms)
100,000	238 ± 2	418 ± 65	87 ± 5	239 ± 26
500,000	637 ± 5	834 ± 85	131 ± 16	277 ± 97
1,000,000	1,124 ± 2	1,311 ± 89	189 ± 146	457 ± 103
2,000,000	2,137 ± 46	2,415 ± 41	277 ± 38	527 ± 90

Table 3: Generation and display time for points. Generation time is the time that is needed to generate the image on the server. Display time is the period starting at the client's request and ending when the response is displayed. The initial time includes query execution and result caching, while regular time only includes reloading the previously cached result. The average $\hat{x}$ and corrected sample standard deviation σ of measured values are reported in the form $\hat{x} \pm \sigma$. The visualized objects are trees, which are represented as points. Listing 3.2 shows the corresponding query.

Table 3 reports times resulting from the query that only returns trees. It shows that the initial display time is larger than the regular display time due to the query execution and result caching. However, even for 2,000,000 trees, the average initial display time is about 2.4 seconds. That is less than the initial display time of the Leaflet marker cluster approach displaying only 100,000 trees instead of 2,000,000.

On average, the regular display time is about a quarter of a second for the query result sizes of 100,000 and 500,000 trees and about half a second for 1,000,000 and 2,000,000 trees. The regular display time of the Leaflet marker cluster approach for 200,000 trees is about the same as the regular display time of our application for 1,000,000 trees. The reported times suggest that our approach is about five to ten times faster than the Leaflet marker cluster approach regarding regular display times. The initial display time of our application is at least an order of magnitude faster than the one of the Leaflet marker cluster.

Table 4 shows the recorded times for the query from Listing 5.1, returning all kinds of objects in Berlin, Germany. The number of coordinates indicates the size of the query result. The reported query results are in the range of 1,000,000 up to almost 17.5 million object-defining coordinates. The initial display time for the reported results is less than five seconds on average for results containing less than 10,000,000 data points. It is less than seven seconds for a result of 17.5 million data points. Regular requests take one second on average for results of about 2.5 million data points and about two seconds for results containing about 10,000,000 data points. The reported times suggest that the initial and the regular generation and display times do not grow more than linear in the size of the query request.

Even though the question of whether these display times can be considered user-friendly overall can not be answered objectively, we can state that our application can at least handle data volumes of the size of several million data points.

		Initial		Regular	
Number of objects	Number of coordinates	generation time (in ms)	display time (in ms)	generation time (in ms)	display time (in ms)
1,000,000	1,000,000	1,437 ± 2	1,647 ± 79	328 ± 58	979 ± 399
1,300,000	2.539.009	2,032 ± 34	2,242 ± 76	541 ± 150	1,099 ± 199
1,500,000	4,457,410	2,573 ± 25	2,810 ± 48	769 ± 208	1,357 ± 419
1,700,000	6,315,668	3,127 ± 27	3,353 ± 64	1,038 ± 309	1,629 ± 236
2,000,000	8,766,993	3,902 ± 39	4,118 ± 65	1,303 ± 202	1,838 ± 336
2,200,000	10,345,885	4,506 ± 25	4,719 ± 70	1,460 ± 127	2,273 ± 198
3,000,000	17,435,660	6,618 ± 23	6,895 ± 79	2,247 ± 157	3,118 ± 496

Table 4: Generation and display time for all kinds of objects. Generation time is the time that is needed to generate the image on the server. Display time is the period starting at the client's request and ending when the response is displayed. The initial time includes query execution and result caching, while regular time only includes reloading the previously cached result. The average $\hat{x}$ and corrected sample standard deviation σ of measured values are reported in the form $\hat{x} \pm \sigma$. The visualized objects are located in Berlin, Germany. They have all kinds of geometries: points, lines and multipolygons. The number of coordinates indicates the number of object-defining coordinates in the query result. Listing 5.1 shows the corresponding query.

6 Conclusion

We face the problem of rendering potentially large amounts of geospatial data in the Web browser. In the paper "Web GIS: Technologies and Its Applications" [2], Alesheikh et al. list different software architectures of *Geographic Information Systems* (Section 2.1), including the thin client approach. Our proposed application is an implementation of the thin client architecture. It differs from other architectures in the component that is processing data. While applications of the thick client architecture process data on the client, applications implementing the thin client architecture perform data processing tasks on the server. The client is just responsible for displaying the computation results. That has the advantage: the computationally demanding data management and processing tasks can be implemented in languages that excel in fast code execution compared to *JavaScript*, the language executed on a Web page. Furthermore, the client's computer resources are no longer a limiting factor.

We implement a Web application highlighting query result data from the query engine *QLever*. The queried data is provided by *OpenStreetMap*, an open-source project on geographic data.

We optimize our application to accelerate the image generation process. Finally, we measure the time it takes to display the requested geospatial information from when it is requested by the user. The results show that our application is capable of visualizing hundreds of thousands of objects in under a second and even millions of objects in seconds.

7 Future Work

There is room for further improvement of our proposed application. Two improvements are listed in the following.

1. Image caching on the frontend could be added as optimization. The images received from the server can be stored with information about the image location and zoom level to be reused when it fits to the visible area. The map can be organized in predefined tiles to facilitate the management of cached images, just as it is the case with the tile layer. The downside is that multiple images may be necessary to visualize the data of one area. However, this optimization ensures that the data of a specific area is only visualized once on every zoom level. That is eliminating redundant requests. Additionally, loading cached images might be faster than requesting the generation of an image.

2. The user interactions zoom and scroll, implemented in our Web application, are very basic. While that is sufficient to give an overview of the location and the shape of the objects present in the query result, other information about the objects, such as name and additional tags, is not considered. Displaying additional object information would be a significant enhancement, at least on zoom levels showing objects in detail. This feature could also be implemented on the server side, adding visual information to the generated image upon request. However, it could also be implemented on the client side. That would require additional object information to be sent to the client.

Bibliography

[1] K.-T. Chang, *Introduction to geographic information systems*. New York: McGraw-Hill Education, ninth edition ed., 2019.

[2] A. Alesheikh, H. Helali, and H. Behroz, "Web GIS: Technologies and Its Applications," 2002.

[3] D. Veil, "Efficient presentation of GeoSPARQL-results," 2021. Bachelor's Thesis.

[4] M. Kulawiak, A. Dawidowicz, and M. E. Pacholczyk, "Analysis of server-side and client-side Web-GIS data processing methods on the example of JTS and JSTS using open data from OSM and geoportal," *Computers & Geosciences*, vol. 129, pp. 26–37, Aug. 2019.

[5] H. Bast and B. Buchhold, "QLever: A query engine for efficient SPARQL+text search," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pp. 647–656, ACM, 2017.

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

[7] J. Bürklin, "QLever UI: A context-sensitive user interface for QLever," 2021. Bachelor's Thesis.

[8] D. Lion, A. Chiu, M. Stumm, and D. Yuan, "Investigating managed language runtime performance: Why JavaScript and python are 8x and 29x slower than c++, yet java and go can be faster?," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, (Carlsbad, CA), pp. 835–852, USENIX Association, July 2022.

[9] E. E. Cureton, "The teacher's corner: Priority correction to "unbiased estimation of the standard deviation"," vol. 22, no. 3, pp. 27–27, 1968. Publisher: Taylor & Francis, eprint: https://doi.org/10.1080/00031305.1968.10480477.