FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Geocluster: Server-side clustering for mapping in Drupal based on Geohash

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Josef Dabernig
Matrikelnummer 0927232

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Prof. Dr. A Min Tjoa
Mitwirkung: Univ.-Ass. Dr. Amin Anjomshoaa

Wien, 30.4.2013

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Josef Dabernig

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30.4.2013

(Unterschrift Verfasser)

Acknowledgement

I'd like to thank everyone who has guided me along writing this thesis. My family, friends, colleagues and teachers for providing so much support. Univ.-Ass. Dr. Amin Anjomshoaa for his great support during the study program and when writing the thesis, as well as Prof. Dr. Silvia Miksch and Prof. Dr. Andrew Frank for their valuable input. Klaus Furtmüllner who helped me combine research, community and work at epiqo. Alex Barth, Felix Delattre, Nedjo Rodgers for introducing me to the Drupal community, as well as Wolfgang Ziegler, Christian Ziegler, Nico Grienauer, Matthias Hutterer, Klaus Purer, Sebastian Siemssen and many others for keeping me there. Théodore Biadala and Nick Veenhof for inspiration, as well as David Smiley, Alex Tulinsky, Chris Calip, Thomas Seidl and Sébastien Gruhier for providing feedback during writing my thesis. Alan Palazzolo, Brandon Morrison, Patrick Hayes, Lev Tsypin, Peter Vanhee amongst many others maintaining Drupal and its mapping modules. Also thanks to the Drupal Association for granting me a scholarship to go to DrupalCon Portland 2013 and talk about mapping with Drupal.

Abstract

This thesis investigates the possibility of creating a server-side clustering solution for mapping in Drupal based on Geohash. Maps visualize data in an intuitive way. Performance and readability of digital mapping applications decreases when displaying large amounts of data. Client-side clustering uses JavaScript to group overlapping items, but server-side clustering is needed when too many items slow down processing and create network bottle necks. The main goals are: implement real-time, server-side clustering for up to 1,000,000 items within 1 second and visualize clusters on an interactive map.

Clustering is the task of grouping unlabeled data in an automated way. Algorithms from cluster analysis are researched in order to create an algorithm for server-side clustering with maps. The proposed algorithm uses Geohash for creating a hierarchical spatial index that supports the clustering process. Geohash is a latitude/longitude geocode system based on the Morton order. Coordinates are encoded as string identifiers with a hierarchical spatial structure. The use of a Geohash-based index allows to significantly reduce the time complexity of the real-time clustering process.

Three implementations of the clustering algorithm are realized as the Geocluster module for the free and open source content management system and framework Drupal. The first algorithm implementation based on PHP, Drupal's scripting language, doesn't scale well. A second, MySQL-based clustering has been tested to scale up to 100,000 items within one second. Finally, clustering using Apache Solr scales beyond 1,000,000 items and satisfies the main research goal of the thesis.

In addition to performance considerations, visualization techniques for putting clusters on a map are researched and evaluated in an exploratory analysis. Map types as well as cluster visualization techniques are presented. The evaluation classifies the stated techniques for cluster visualization on maps and provides a foundation for evaluating the visual aspects of the Geocluster implementation.

Kurzfassung

Diese Diplomarbeit erforscht die technische Möglichkeit zur Erstellung einer Geohash-basierten, server-seitigen Cluster-Lösung für Kartenanwendungen in Drupal. Landkarten visualisieren Daten auf intuitive Weise. Die Performanz und Lesbarkeit von digitalen Kartenanwendungen nimmt jedoch ab, sobald umfangreiche Datenmengen dargestellt werden. Mittels JavaScript gruppiert client-seitiges Clustering überlappende Punkte zwecks Lesbarkeit, ab einer gewissen Datenmenge verlangsamt sich jedoch die Verarbeitungsgeschwindigkeit und die Netzwerkverbindung stellt einen Flaschenhals dar. Die zentrale Forschungsfrage ist daher: Implementierung von server-seitigem Clustering in Echtzeit für bis zu 1.000.000 Punkten innerhalb einer Sekunde und die Visualisierung von Clustern auf einer interaktiven Karte.

Clustering ist ein Verfahren zur automatisierten Gruppierung in Datenbeständen. Algorithmen der Clusteranalyse werden evaluiert um einen geeigneten Algorithmus für das server-seitige Clustering auf Landkarten zu schaffen. Der entworfene Algorithmus nutzt Geohash zur Erstellung eines hierarchischen Spatialindex, welcher das Clustering unterstützt. Geohash kodiert Latitude/Longitude Koordinaten in Zeichenketten mit räumlich-hierarchischer Struktur unter Beihilfe der Morton-Kurve. Durch Einsatz des Geohash-basierten Index kann die Zeitkomplexität des Echtzeit-Clusterings drastisch reduziert werden.

3 Varianten des Clustering Algorithmus wurden als Geocluster Modul für das Content Management System und Framework Drupal implementiert. Die erste, PHP-basierte Variante skaliert nicht. Die zweite Variante mittels MySQL konnte in Tests bis 100.000 Punkte unter 1 Sekunde clustern. Schlussendlich skaliert das Clustering mit Apache Solr bis über 1.000.000 Elemente und erfüllt somit das primäre Forschungsziel.

Neben der Performance-Analyse wurden auch Techniken zur Visualisierung von Clustern auf Karten erforscht und im Rahmen einer explorativen Studie verglichen. Ver-

schiedene Kartentypen, als auch Visualisierungsformen von Clustern werden präsentiert. Die Evaluierung klassifiziert die Techniken zwecks der Darstellung auf Karten und bildet somit die Grundlage für die Diskussion der Geocluster-Implementierung.

Contents

Chapter 1

Introduction

1.1 Motivation

Digital mapping applications on the Internet are strongly emerging. Big players like Google Maps[1] and OpenStreetMap[2] provide online maps, that users can view and interact with.

Maps allow telling stories and communicating data in a visual way. Using open source tools such as TileMill[3] and online services like CloudMade[4], more and more people are able to create their own custom maps. The Free and Open Source content management system and framework Drupal[5] provides tools for adding, editing and visualizing geographic data on maps. This allows to integrate interactive map applications into web sites.

Maps provide a birds-eye-view on the geography and information captured in such system. Sometimes, they are used to get a quick overview of points-of-interest in a certain area. If a large amount of information is contained in such an area, problems in terms of computability and visual clutter arise: visualizing thousands of points on a single map both challenge human and computers. Obviously, when telling a story, information needs to be told in a compact way as the human brain can only process a limited amount of data at the same time. Similarly, large amounts of data involve a higher burden on the computer components that participate in the web mapping process [**?**, **?**].

[1] https://maps.google.at
[2] http://www.openstreetmap.org/
[3] http://mapbox.com/tilemill
[4] http://cloudmade.com/
[5] http://drupal.org

Clustering[6] is a technique for grouping objects with similarities that can be used to reduce visual clutter as described before. Various client-side Javascript libraries like Leaflet.markercluster[7] exist for clustering points on maps. This enhances performance and readability of data-heavy map applications. But still, all data needs to be transferred to the client and processed on a potentially slower end user device. By clustering data on the server-side, the load is shifted from the client to the server which allows displaying larger amounts of data in a performant way. Professional services like maptimize[8] provide such a functionality, while in the open source space little libraries and frameworks exist for server-side clustering of geospatial data.

In order to create interactive maps based on large data sets, this thesis evaluates and implements a performance-optimized server-side clustering algorithm for Drupal.

1.2 Outline of the thesis

Chapter **??** introduces the technical foundations for this thesis. It provides an overview by explaining cluster theory and comparing four main clustering algorithms. Further, a discussion on spatial data computation introduces space order and space decomposition methods, Quadtrees and the Geohash encoding. An overview of foundational concepts in web mapping as coordinate systems, map projections and spatial data types is provided. Finally, basic concepts of geovisualization as visual variables, visual data exploration techniques and clutter reduction are explained.

Chapter **??** discusses the state of the art of related technologies for the thesis. An explanation of a modern web mapping stack is given as well as the basics of Drupal & mapping technologies. Further, the state of the art for client-side and server-side clustering technologies in web mapping is analyzed. The chapter is concluded by a section on visual mapping that list map visualization types, as well as cluster visualization techniques and summarizes them in an evaluation.

Chapter **??** states the objectives and exemplary use cases for the thesis. It defines the goals to be accomplished as a result of the implementation part of the thesis.

Chapter **??** describes the realization of the server-side clustering algorithm. First, an analysis based on the objectives stated in the previous chapter is given. Subsequently, the Geohash-based clustering algorithm is defined. Finally, the architecture and implementation of the algorithm for Drupal is explained in detail.

[6] http://en.wikipedia.org/wiki/Cluster_analysis
[7] https://github.com/Leaflet/Leaflet.markercluster
[8] http://www.maptimize.com

Chapter **??** discusses the implementation of use cases for the realized server-side clustering algorithm.

Chapter **??** finally evaluates the results of the thesis. It contains performance tests and a visual evaluation on how the defined objectives have been accomplished. Final conclusions are made and an outlook on future work is given.

CHAPTER 2

Foundations

In the following chapter, the technical foundations of cluster theory, computing spatial data and fundamental concepts of web mapping are explained. The definition of the clustering task, an overview cluster analysis history as well as basic definitions of cluster types, clustering techniques and proximity measures are given. Four foundational clustering algorithms are explained to provide an overview and understanding of the basic differentiation between clustering techniques. Approaches for dealing with spatial data are laid out by discussing space order methods, space decomposition methods, Quadtrees and the Geohash encoding.

2.1 Clustering

Clustering is the task of grouping unlabeled data in an automated way. It can also be described as the unsupervised classification of patterns into groups. The techniques of cluster analysis are applied in numerous scenarios including data mining, document retrieval, image segmentation and pattern classification. They are used to solve different tasks including pattern-analysis, grouping, decision-making or machine-learning [**?**].

Cluster analysis has been studied since the early beginnings of computer science and applies to a broad number of research fields. Different research communities have created a variety of vocabularies to describe methods related to clustering. The following terms are used in literature as synonyms for the term cluster analysis: Q-analysis, topology, grouping, comping, classification, numerical taxonomy and unsupervised pattern recognition.

As indicated, clustering is a wide and generic term. Often, it is used to refer to specific concepts which are appropriate for solving specific tasks. This means, that efficient

clustering algorithms have been developed and studied over the years for certain research fields. While such algorithms might perform well under certain circumstances, they might be completely inappropriate for other use cases. Imagine, an algorithm that fits image segmentation well but is less useful in machine-learning [**?**, **?**].

Clustering is a general concept that applies to multivariate data. Spatial data in this sense is a special case, and 2-dimensional spatial data reduces the problem space even further to planar space. Figure **??** visualizes such an example of clusters of point patterns in two dimensional space. Humans can understand and perform this kind of clustering tasks in an efficient way, whereas clustering in high-dimensional space is difficult for humans to obtain.

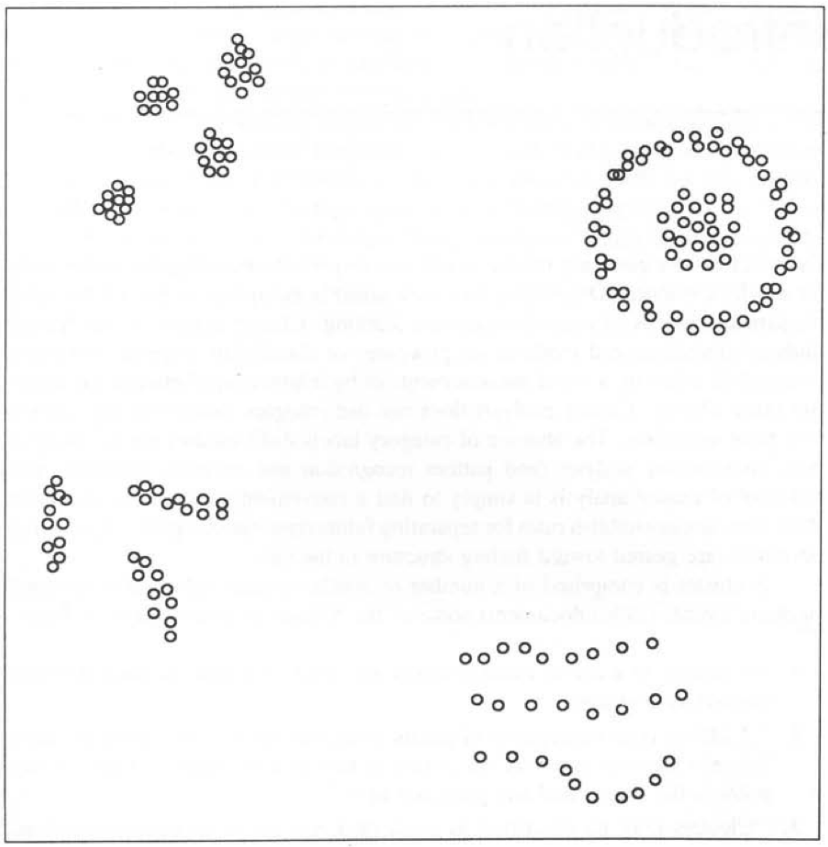


Figure 2.1: Clusters of point patterns in two dimensions [**?**, p 2].

2.1.1 The Clustering task

Clustering is the task of aggregating items (also described as features) into clusters, based on similarities or proximity. A.K. Jain, M.N. Murty and P.J. Flynn [**?**] define the following steps involved in a typical pattern clustering activity:

1. pattern representation (optionally including feature extraction and/or selection),
2. definition of a pattern proximity measure appropriate to the data domain,
3. clustering or grouping,
4. data abstraction (if needed), and
5. assessment of output (if needed).

2.1.2 History

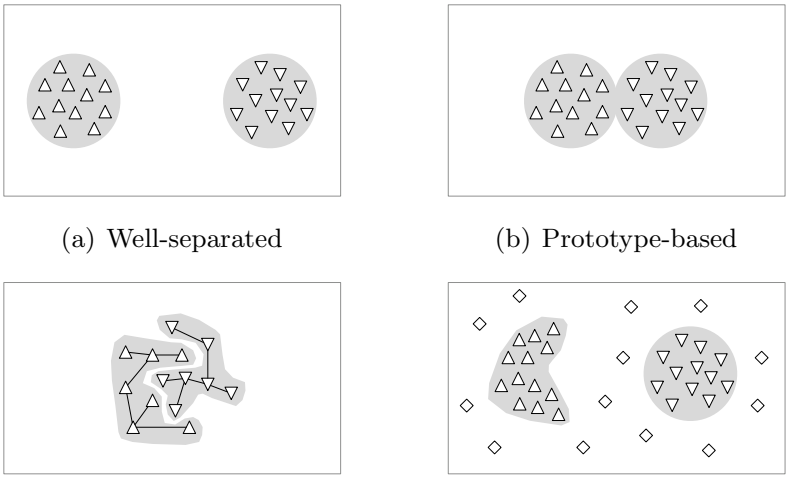
K-means, one of the oldest and most widely used clustering algorithms was introduced already in 1967 [**?**, **?**]. Tryon and Bailey (1970) wrote one of the first books on cluster analysis. In 1973, Anderberg published "Cluster analysis for applications", a book that Jain and Dubes describe as "the most comprehensive book for those who want to use cluster analysis" [**?**]. While Tryon and Bailey focus on a single clustering approach (BC TRY), Anderberg already gives a comprehensive overview of clustering methods, strategies and a comparative evaluation of cluster analysis methods.

Clustering algorithms were improved and developed further over time, i.e. to account for performance issues. Prominent algorithms in that area include CLARANS [**?**] and BIRCH [**?**] - they have a time complexity linear to the number of patterns. A popular, density based clustering algorithm is DBSCAN [**?**]. Jain and Dubes summarize hierarchical and partitional clustering approaches in "Algorithms for Clustering Data" (1988) with a special focus on applications in image processing. Subsequent publications on cluster analysis are released continuously [**?**].

2.1.3 Cluster types

Clusters are groupings of similar objects. Different models of interpretation for clusters exist: most notably, they can be classified into different types of clusters as illustrated in figure **??**. These type are:

- **Well-separated** clusters have the property that objects within a cluster are closer to each other than any object outside of the cluster. As the name suggests, this is only possible when the data contains natural clusters that are quite far from each other.
- **Prototype-based** clusters are defined, so that objects are closer to their cluster's prototype than to any other one's. Prototypes of clusters are either centroids (the mean of all points for a cluster) for continuous data or medoids (the most central point within a cluster) for categorical data.
- **Graph-based** clusters can be defined as *connected components* within a graph. That is a group of objects (nodes), where the objects are connected to one another but disconnected from objects outside of the cluster.
- **Density-based** clusters group objects within dense regions that are surrounded by a region of low density. Such a definition is often employed when noise is present or clusters are irregularly shaped [**?**].



(a) Well-separated (b) Prototype-based

(c) Graph-based (d) Density-based

Figure 2.2: Types of clusters: (a) Well-separated, (b) Prototype-based, (c) Graph-based, (d) Density-based [**?**, p 9].

2.1.4 Clustering techniques

Literature research reveals classifications of clustering techniques according to various aspects. Jain, Murty and Flynn primarily group the algorithms into hierarchical and partitional ones, see figure **??** [**?**]. On the other hand, Stein and Busch split them into hierarchical, iterative, density-based and meta-search-controlled, see figure **??** [**?**]. The differentiation of properties into groupings of clustering techniques and cross-cutting aspects is inconsistent among publications. This again shows the wide variety in which cluster analysis is being discussed and developed.

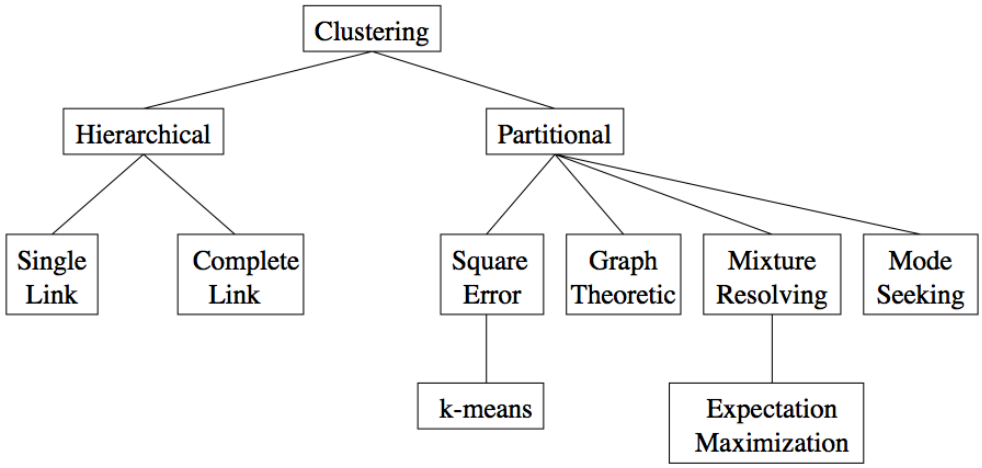


Figure 2.3: A taxonomy of clustering approaches. [**?**, p 275].

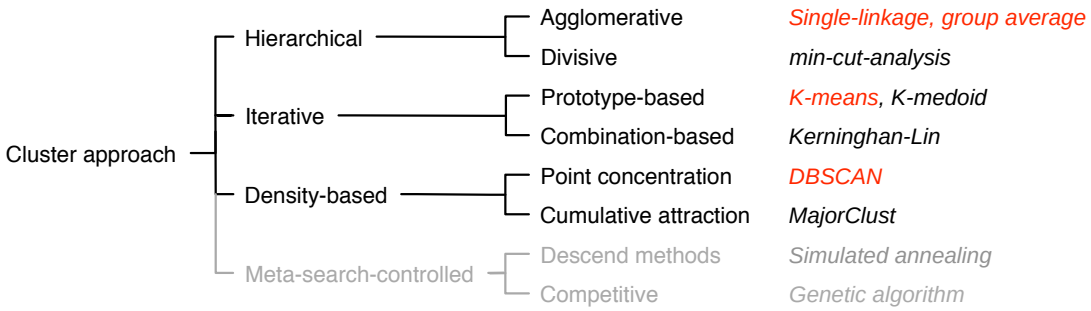


Figure 2.4: A taxonomy of cluster algorithms as cited in [**?**, p 14], based on [**?**].

In the following, we will discuss aspects of cluster theory which are used to differentiate clustering techniques:

- **Hierarchical versus Partitional**. Whether a clustering is nested, is the classic differentiation between clustering techniques. *Partitional clustering* divides the data into a single set of non-overlapping clusters. It usually is driven by an *iterative* approach that optimizes the result. *Hierarchical clustering* organizes clusters as a tree. Each node in the tree is the union of its children. Both clustering types are related: applying partitional clusterings in a sequence can lead to a hierarchical clustering and cutting the hierarchical tree at a particular level produces a partitional clustering [**?**].

  An example of the relationship between hierarchical and partitional clustering is given by the following example of density-based algorithms. *DBSCAN* produces simple data partitions and was further developed into *OPTICS* which clusters data hierarchically [**?**].

- **Agglomerative versus divisive**. *Agglomerative clustering* algorithms start with single items and successively merge them together into clusters. On the other hand, *divisive clustering* algorithms begin with a single cluster that contains all items and splits it up until a stopping criterion is met. Each such merging or splitting procedure can be seen as one level in the hierarchical clustering tree [**?**].

- **Hard versus Fuzzy**. *Hard clustering* algorithms assign every item to a single cluster, this means that the clustering is *exclusive*. A *fuzzy clustering* algorithm may attribute an item to multiple clusters in a *non-exclusive way* by assigning degrees of membership. A fuzzy clustering may be converted into a hard clustering by assigning every data item to the cluster with the highest degree of membership [**?**, **?**].

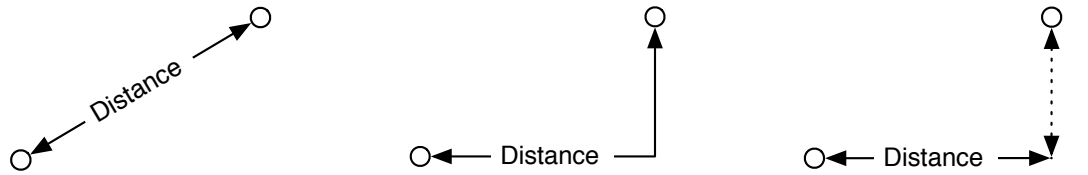
- **Complete versus Partial**. With *complete clustering*, assigning every point to a cluster is required. *Partial clustering* relaxes this requirement so that not every point needs to be assigned to a cluster. This can be particularly useful when clustering data sets with outliers and noise. In such cases, the partial clustering can be used to focus on crowded areas [**?**].

Further aspects include *monothetic versus polythetic* and *incremental versus non-incremental* clustering techniques.

2.1.5 Proximity

Similarity is fundamental to the definition of a cluster. In order to measure similarity between clusters, clustering algorithms evaluate the proximity. Continuous data requires different proximity measurements than categorical data.

For continuos data, the proximity between items is typically quantified by dissimilarity in terms of distance measures. The three main distance functions are visualized in figure **??** and explained as follows [**?**]. x and y represent two data items that are part of the clustering process. Their spatial coordinate on each axis is accessed using the index i.



(a) Eucledian distance (b) Manhattan distance (c) Chebychev distance

Figure 2.5: Distance measures for continuous data [**?**, p 12].

- **Euclidian distance**. The euclidian distance is the geometric distance in the n-dimensional space. It is is the most common type of distance and defined as

$$distance(x,y) = \sqrt{\sum_i (x_i - y_i)^2}$$

- **Manhattan distance**. The manhattan distance, also known as city-block distance is distance when walking from one point to another following the axes. Compare with walking by following a raster like in manhattan. It is defined as

$$distance(x,y) = \sum_i |x_i - y_i|$$

- **Chebychev distance**. This distance measure returns the maximum distance between two points on any dimensions. It may be appropriate where two items are defined as 'different', if they are different on any one of the dimensions.

$$distance(x,y) = max|x_i - y_i|$$

2.2 Clustering algorithms

Researchers have created a multitude of algorithms, each appropriate for a certain task. As we will find out later, the requirements to the spatial clustering algorithm for this thesis are specific, which out-rules most scientific clustering algorithms which are geared

towards image recognition or other disciplines. To provide an overview and to understand the basic differentiation of clustering techniques explained in **??**, in this chapter we will introduce 4 foundational clustering algorithms:

- K-means (Squared Error Algorithm)
- Agglomerative Hierarchical Clustering Algorithm
- DBSCAN (Density-based)
- STING (Grid-based)

2.2.1 Squared Error Algorithms: K-means

The K-means is the most commonly used and simple algorithm based on a *squared error criterion*. It creates a one-level partitioning of the data items by dividing them into K clusters. By starting with a random initial partition, it iteratively reassigns the patterns to clusters based on similarity. The clustering process is completed when a convergence criterion is met, i.e. no further reassignments happen. Clusters are defined by *cluster prototypes*: the centroid of the clustered items. Alternatively, the *K-medoid* algorithm uses the most representative data item instead of the centroid.

The time complexity of the K-means algorithm is linear to the number of points:

$$time\ complexity = \mathrm{O}(n)$$

input: the number of clusters, K

```
1 Select K points as initial centroids;
2 while Centroids do change do
3 |   Form K clusters by assigning each point to its closest centroid;
4 |   Recompute the centroid of each cluster;
5 end
```

Algorithm 2.1: K-means algorithm [**?**]

The selection of initial centroids affects the final outcome of the clustering process. Choosing them randomly is a simple, but not very affective approach. This can be compensated by applying multiple runs of the algorithm, to retrieve an optimal result set. Optimizations to the centroid initialization include applying a hierarchical clustering or selecting distant points in the beginning.

Assigning points to the closest centroid requires a proximity measure, as explained in **??**. Simple measures like the Euclidian distance are preferred, as this step needs to

happen repeatedly within the algorithm. For each cluster, the centroid needs to be recalculated afterwards. This procedure is repeated until centroids do not change any more [**?**, **?**].

Algorithm **??** and figure **??** illustrates the K-means clustering process.

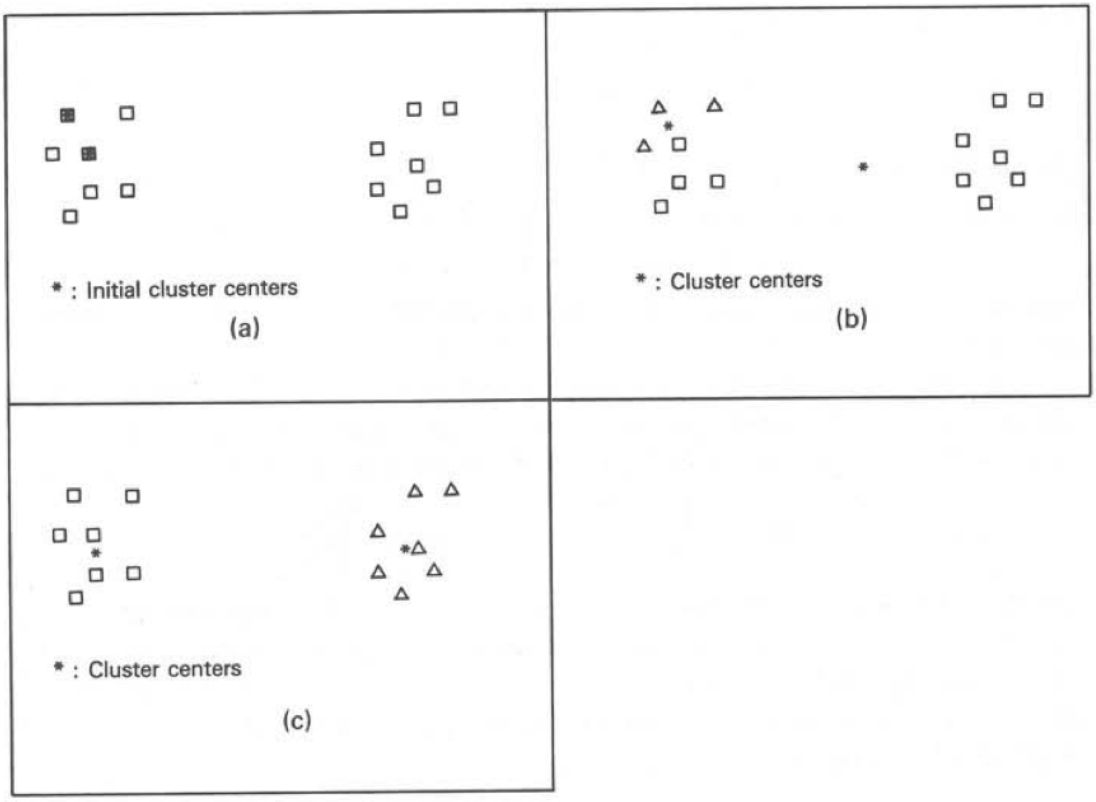


Figure 2.6: Convergence of K-means clustering: (a) initial data; (b) cluster membership after first loop; (c) cluster membership after second loop. The different shapes of items represent their cluster membership. [**?**, p 99].

2.2.2 Agglomerative Hierarchical Clustering Algorithm

This exemplary, *hierarchical clustering algorithm* creates a hierarchy of nested sub clusters by serial partitioning. Its *agglomerative* nature makes it start with every data item as a single cluster and merge them sub-sequentially. As an alternative, a *divisive* hierarchical clustering would start with one cluster containing all data points and recursively split them up. At each level, a partitioning can be extracted, for example to serve as initial set of centroids for the previously discussed K-means algorithm.

The time complexity of the agglomerative hierarchical clustering algorithm is:

$$\textit{time complexity} = \mathrm{O}(n^3)$$

At the beginning of the agglomerative hierarchical clustering algorithm, each point is assigned to its own cluster. A proximity matrix is calculated to store the distances for all pairs of data items based on the chosen proximity measure.

To merge the two closest clusters, different heuristics may be applied. Most importantly, *single-link* hierarchical clustering algorithms measure the distance between two clusters by the *minimum* distance between all pairs of items from the two clusters. In contrast, *complete-link* algorithms use the *maximum* distance to create compact clusters and prevent chaining effects. Other approaches are based on *average linkage* or *Ward's method*. After merging the clusters, the proximity matrix will be updated, so that it reflects the current state of the clustering process. This procedure is repeated until all clusters have been merged, each step in the loop yields a level in the hierarchical clustering [**?**, **?**, **?**].

Algorithm **??** describes agglomerative hierarchical clustering and figure **??** illustrates the clustering process visually as a dendrogram.

```
1 Assign each point to its individual cluster;
2 Compute the proximity matrix;
3 while Number of clusters is larger than one do
4 |   Merge the closest two clusters;
5 |   Update the proximity matrix to reflect the proximity between the new cluster and
  |   the original clusters;
6 end
```

Algorithm 2.2: Agglomerative hierarchic algorithm [**?**]

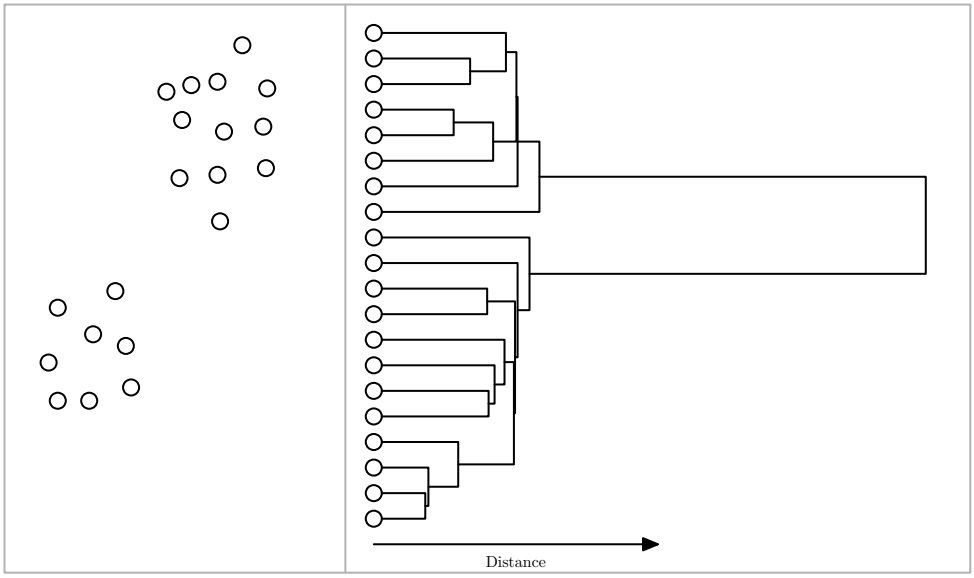


Figure 2.7: Dendrogram [**?**, p 20].

2.2.3 Density-based clustering algorithms: DBSCAN

Density-based algorithms cluster regions of high density and separate them from regions with lower density. The density-based approach is different to the previously discussed distance-based methods. Those tend to perform well in the detection of spherical-shaped clusters, but discovering arbitrary shapes is a problem. Density-based algorithms overcome this limitation and are also insensitive to noise points, so that outliers get isolated.

```
1 while Point is unclassified do
2 |   Find points within region ϵ;
3 |   if number of points within region > MinPts then
4 |   |   Start new cluster with Point;
5 |   |   Search regions of points in new cluster and expand cluster;
6 |   end
7 end
```

Algorithm 2.3: DBSCAN algorithm [**?**]

DBSCAN starts with an unclassified point. Subsequently, the density of the point is calculated by computing all its neighborhood points within a radius ϵ. Based on the density, the algorithm classifies points as *core points*, *border points* or *noise points*:

- *core points* have a number of points within their neighborhood that exceeds the threshold defined as $MinPts$
- *border points* don't match the previous criterion but they themselves fall into the neighborhood of another core point
- *noise points* are outside of any neighborhood and therefore are neither core points nor border points

Any core point will be expanded to a cluster in the procedure until every point has been classified [**?**, **?**].

The quadratic time complexity of DBSCAN $O(n^2)$ can be optimized by using an indexing structure for the neighborhood queries, resulting in a logarithmic time complexity: [**?**]

$$time\ complexity = O(n \log n)$$

Algorithm **??** and figure **??** illustrate the DBSCAN clustering process.

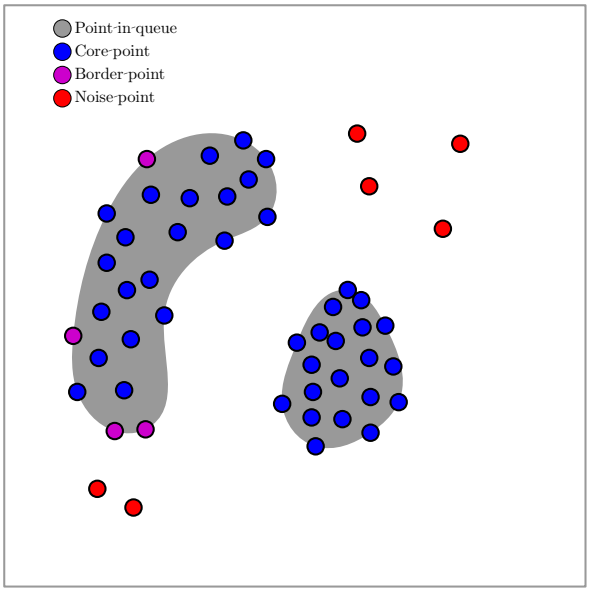


Figure 2.8: DBSCAN algorithm [**?**, p 26].

2.2.4 Grid-based algorithms: STING

The time complexity of the previously discussed algorithms is at least linear to the number of points that have to be clustered. Grid-based algorithms overcome this performance limitation by partitioning data to be clustered in a grid structure. The clustering process is executed on pre clustered cells of the grid structure, which obviously scales better.

The STING algorithm takes such a grid-based approach and divides the data points into a grid of rectangular cells. The cells form a hierarchical structure, so that different levels of grids correspond to different resolutions. Every cell in the grid is sub-divided into a further partitioning one level deeper in the hierarchy. STING therefore precomputes a hierarchical index with various levels of granularity on which the clustering process is later executed. The name-giving property of STING (Statistical INformation Grid) is that each cell contains statistical information which can be used to answer queries.

The time complexity can be shifted to the computation of the grid which is linear to the points of data O(n). The query processing time is reduced to O(g), where g is the constant of number of cells at the bottom level of the computed hierarchy. As per the construction of the hierarchy, it can be assumed that $g \ll n$ [**?**, **?**].

Figure **??** illustrates a hierarchical grid used in the STING clustering process.

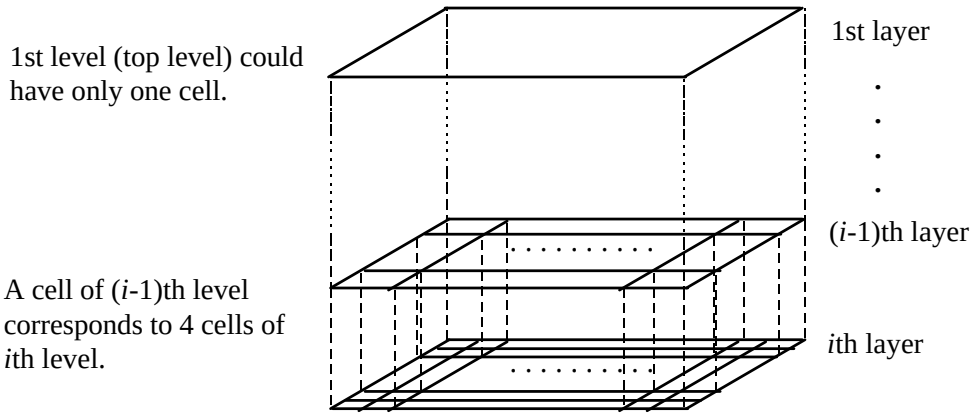


Figure 2.9: Hierarchical Structure of the STING algorithm [**?**, p 5].

2.3 Spatial data

A central aspect in web mapping is dealing with spatial data. Decisions on how to structure and store spatial data highly influence the computation tasks that may be performed on such data. This mainly depends on the type of spatial data, where points are the most basic ones. Depending on the use case, different types of data are to be represented and stored: points, lines, rectangles, regions, surfaces and/or volumes. While most of the discussed concepts may be extended or generalized for processing more complex types of data, this is out of scope for this thesis. Instead, we focus on points as the most common type of spatial data [**?**].

One fundamental way to store spatial data is the quadtree, a hierarchical data structure based on recursive decomposition of space. Hanan Samet attributes the history of quadtrees (and octrees which are their 3-dimensional extension) to Dijkstra, who invented a one-level decomposition of a matrix into square blocks. Morton then applied this technique to creating a spatial index (z-order). We will first discuss space orders & decomposition and then put those into context when outlining the basics of the quadtree and Geohash.

2.3.1 Space order methods

In order to store multi-dimensional, spatial data in a sequential storage like computer memory, the data needs to be serialized. Consider a pixel image as an example. Its 2-dimensional pixel values are positioned in planar space. In order to store the image, those pixels have to be processed in a predefined order, such that they can be serialized into a 1-dimensional array of memory units.

The traditional order for storing a raster of image data was row by row. The row order sequentially processes the raster row by row from left to right, starting at the top left corner. In contrast, the row-prime order switches the horizontal processing direction at the end of each row. This leads to a higher locality as it has the property of always moving to a 4-neighbor [**?**].

Besides the discussed row orders, additional space-ordering methods have been developed to serve different purposes. The Morton and Peano-Hilbert orders both visit entire sub quadrants first, before exiting them. The Morton order is easier to compute, because the position (key) of each element in the ordering can be determined by interleaving the bits of the x and y coordinates of that element. One disadvantage of the Morton order are the gaps: the longest move in a raster of 2^n by 2^n is one column and $2^n - 1$ rows (or vice versa). A better locality is offered by the Hilbert-Peano order which always has the property of moving to a 4-neighbor. This advantage on the other hand has the cost of a more complex definition. Calculating keys for the Hilbert-Peano order is more difficult.

The higher complexity of the Hilbert-Peano order obviously shows that its recursion is harder to define as well. Figure **??** illustrates these fundamental space orders with two further ones that allow for ordering unbounded space in two (Cantor-diagonal order) or four directions (spiral order) [**?**].

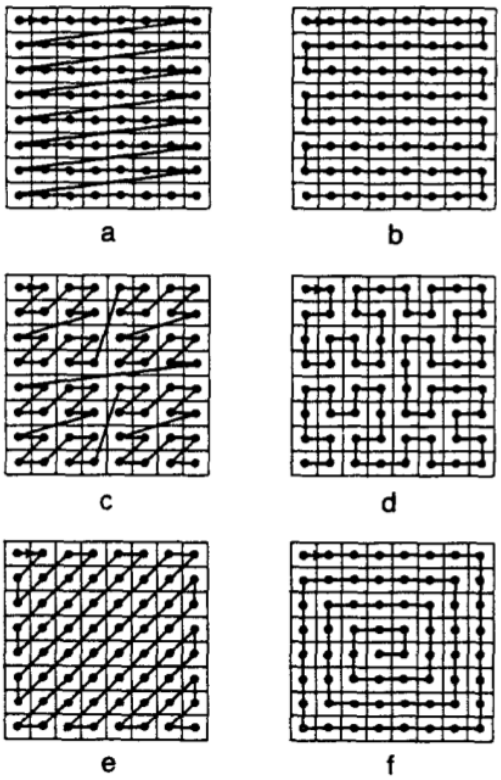


Figure 2.10: The result of applying a number of different space-ordering methods to an 8×8 image whose first element is in the upper left corner of the image: (a) row order, (b) row-prime order, (c) Morton order, (d) Peano-Hilbert order, (e) Cantor-diagonal order, (f) spiral order [**?**, p 14].

The fact, that the Hilbert-Peano order has the property of always moving to a 4-neighbor shouldn't be misinterpreted, as still there are gaps. "A bijective mapping from multidimensional data to one dimension cannot be done the way that in any case nearby multidimensional points are also close together in one dimension [**?**]". Figure **??** clearly shows what Samet describes as "both the Morton and Peano-Hilbert order exhaust a quadrant or subquadrant of a square image before exiting it [**?**]". This means, that the orders maintain locality for those quadrants based on the hierarchy, but the edges are still disconnected. The same issue applies to Geohash as explained in chapter **??**.

2.3.2 Space decomposition methods

By definition, space decomposition methods partition space in a way so that,

- partitions are infinitely repetitive patterns for images of any size,
- partitions are infinitely decomposable to generate finer sub-partitions of higher resolution. [**?**]

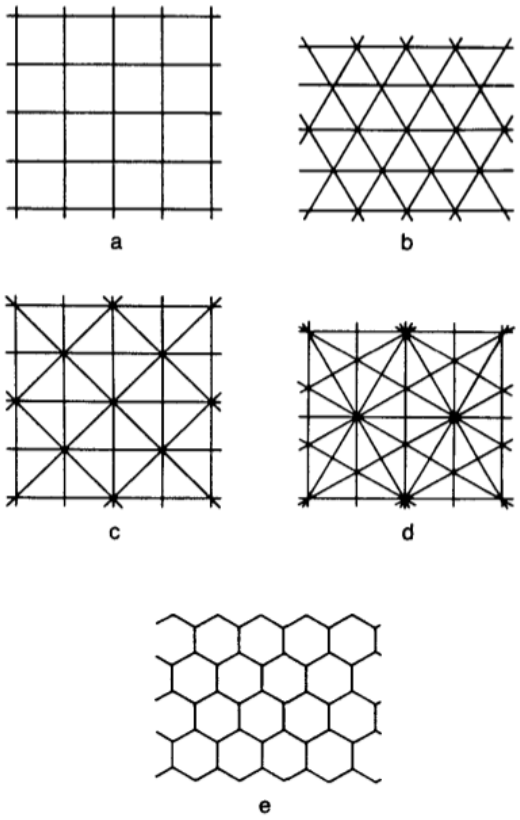


Figure 2.11: Sample tessellations: (a) [4^4] square; (b) [6^3] equilateral triangle; (c) [4.8^2] isoceles triangle; (d) [4.6.12] 30-60 right triangle; (e) [3^6] hexagon [**?**, p 17].

Various space decomposition methods exist. They can be classified depending on the shapes that are used for the partitioning patterns. Polygonal shapes are computationally simpler and can be used to approximate the interior of a region while non-polygonal shapes are more geared towards approximating the perimeter of region. Figure **??** visualizes a number of basic, polygon-based space decomposition methods.

The simplest polygonal space decomposition method is based on squares. It is directly related to the Morton and Peano-Hilbert space order methods as described in the previous chapter **??**. Both orders work in a hierarchical manner and visit entire sub-quadrants first, before continuing further. This is why they are predestined to decomposing space into squares as indicated in figures **??** and **??**.

2.3.3 Quadtree

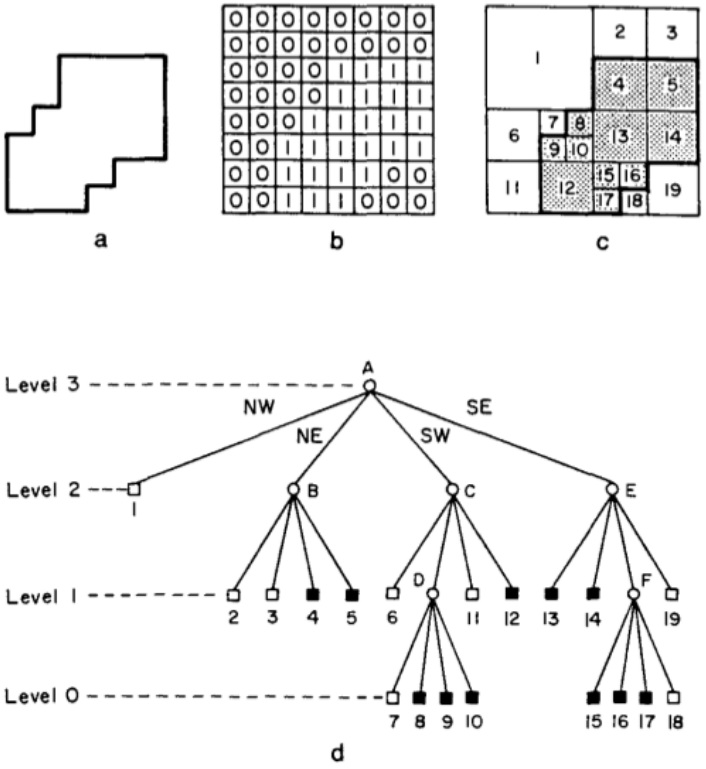


Figure 2.12: An example of (a) a region, (b) its binary array, (c) its maximal blocks (blocks in the region are shaded), and (d) the corresponding quadtree [**?**, p 3].

Quadtrees are hierarchical data structures based on recursive decomposition of space. Their original motivation was to optimize storage by aggregating identical or similar values. Over time, they have also been studied to optimize execution time of spatial application and have been established as a common practice for representing and storing spatial data. The resolution of a quadtree can be fixed or variable. It directly relates to the number of decomposition times of the space where the data points live. A standard-

ized implementation is the region quadtree which subdivides space into four equal-sized quadrants.

Quadtrees can be constructed in different ways. A top-down approach implements the Morton order which maps the multidimensional data to one dimension and has been introduced in chapter **??**. Also note that the term quadtree has taken various meanings: actually it is a trie (or digital tree) structure because each data key is a sequence of characters. "A node at depth i in the trie represents an M-way branch depending on the i-th character" [**?**].

2.3.4 Geohash

Geohash is a latitude/longitude geocode system based on the Morton order, described in **??**. It encodes geographic point coordinates into string identifiers that reflect a hierarchical spatial structure. By having a gradual precision degradation property, two Geohashes that share a common prefix imply proximity described by the length of the shared prefix [**?**, **?**].

Originally, the Geohash encoding algorithm has been developed and put into the public domain by Gustavo Niemeyer when creating the web service geohash.org. The service allows to encode spatial coordinates into unique string identifiers and vice versa [**?**]. Amongst other applications, it has also been incorporated into geospatial search plug-ins of the Apache Solr search platform [**?**] and leveraged for real-time, location-aware collaborative filtering of web content by HP Labs [**?**].

A Geohash is constructed by interleaving the bits of the latitude and longitude values and converting them into a string of characters using a Base 32 encoding. As every Base 32 symbol is represented by an uneven number of 5 bits, the resulting space decomposition is a rectangular grid formed by 4×8 or 8×4 cells. The orientation of the resulting rectangles alternates between vertical for characters of an odd index and horizontal for such characters that are positioned at an even index within the Geohash. Figure **??** illustrates how the first character of a Geohash string splits the projected earth into an 8×4 grid of horizontally aligned rectangles [**?**, **?**].

The fact that points with a common prefix are near each other must not be confused with the converse. Due to the nature of the Morton order, edge cases exist. Two points may be very close to each other, without sharing an equally long prefix. Figure **??** illustrates an example of two closely positioned points being located within different Geohashes. The first point being at the very lower end of the "DRT" Geohash cell and the second point positioned closely at the upper end of the "DRM" Geohash cell.

Figure 2.13: Space decomposition of the geohash algorithm on the first level [**?**].

2.4 Web Mapping

Maps have become an almost instinctive way of seeing our world. They probably first appeared over 18,000 years ago and already in the 16th century, they were produced in large numbers for navigational and military purposes. Maps are powerful tools that help organize boundaries and administrative activities. They allow telling stories, visualizing data and understanding geographic contexts [**?**].

Recently, Google Maps[1] has made digital maps available to a large number of internet users. *Digital natives* are used to navigate by using interactive maps on their smart phones and look up places on online maps on their computers.

Web mapping describes the whole process of designing, implementing, generating and delivering maps on the internet. It applies theoretical foundations from web cartography to the technical possibilities and boundaries of constantly evolving web technologies. The continuous development of related technologies has created a wide variety of *types*

[1] http://maps.google.com

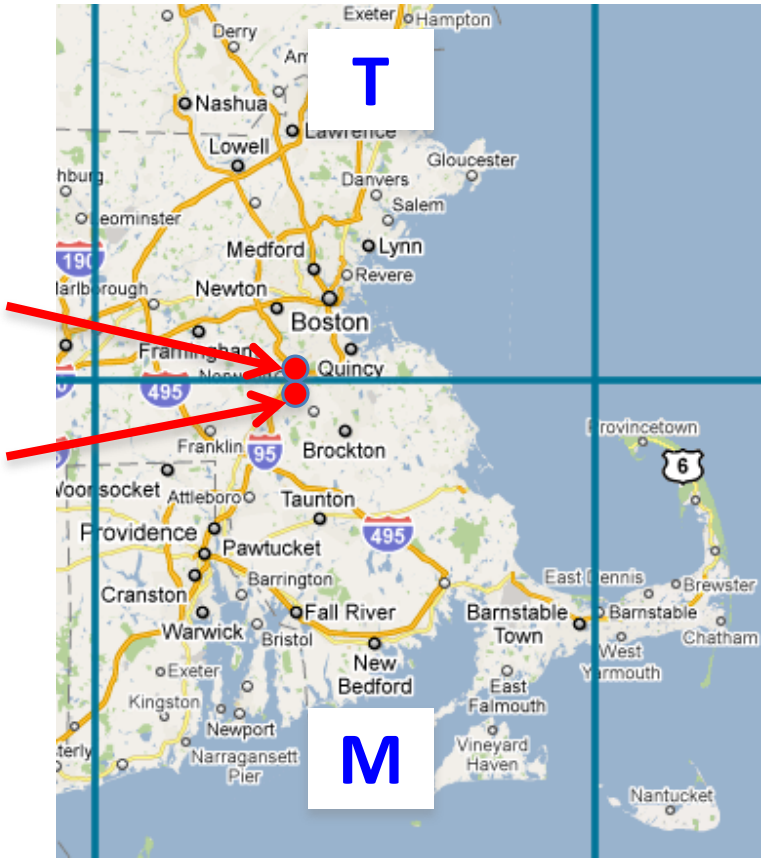


Figure 2.14: Geohash edge case, where two closely positioned points do not share the same Geohash prefix [**?**].

of web maps: from analytic, animated, collaborative and dynamically created web maps to online atlases, realtime and static web maps [**?**].

In order to represent spatial locations, *reference systems* are used, that subdivide the geographic area into units of common shape and size. Such spatial reference systems consist of a *coordinate system*, a *datum* and a *projection*. *Geodetic datums* are models that approximate the shape of the Earth. In the following two chapters, the remaining concepts of coordinate systems and map projections will be explained in more detail.

2.4.1 Coordinate systems

In mapping, a coordinate system is used to represent spatial locations in a numeric way. We mainly differentiate between *Cartesian* and *Ellipsoidal* coordinate systems.

- **Cartesian coordinate systems** express a spatial location by specifying the distances from a point of origin on axes. The axes are usually labeled X, Y and Z for locations in three-dimensional space. As an example, the *Earth Centered, Earth Fixed X, Y, and Z (ECEF)* coordinate system is used by positioning technologies such as GPS. The coordinates of New York in ECEF are:

$$(X, Y, Z) = (1334.409\ km, -4653.636\ km, 4138.626\ km)$$

  "Earth centered" emphasizes that the origin of the axes is defined to be at the geocenter of the planet. For many tasks, this system isn't intuitive as the values don't indicate, if a location is on, above or below the surface of the Earth.

- **Ellipsoidal coordinate systems** describe a more convenient way of expressing spatial location on the Earth's surface. A *reference ellipsoid* approximates the shape of the Earth by an equatorial and a polar radius. As a result, positions at the surface of the Earth can be represented as angles. This defines the primary way of expressing coordinates as a pair of *latitude* and *longitude* values.

  - **Latitude** classifies the angular distance towards north and south from the equator which is at 0°. Positive latitude values represent the northern hemisphere up the the pole at 90°. Negative values are located below the equator where the south pole marks the lower limit at −90°.
  - **Longitude** denotes the angular distance towards west and east. It's zero-mark is a latitude of 0°, which runs north-south through the Royal Observatory at Greenwich in the UK. In contrast to latitude values, the longitude encloses a whole circle around the earth. Negative values down to −180° are located west and positive values up to 180° are positioned east of Greenwhich.

  In classic mapping applications, latitude and longitude values are measured in *degrees, minutes and seconds* of the sphere. New York City is located at 40° 43′ 0″ North, 74° 0′ 0″ West. For computational purposes, web mapping is largely based on a *decimal degree* representation of such values. The equivalent decimal degree value for New York in that case is the pair of

$$(latitude, longitude) = (40.716667, -74)$$

  A common pitfall in web mapping is mixing up the order of latitude and longitude values. Having latitude before longitude is the standard, which means to state the vertical position before the horizontal. This contradicts with classic Cartesian x,y coordinate systems and often leads to confusion. Some mapping APIs expect latitude first, while others are designed to begin with longitude values [**?**, **?**].

2.4.2 Map Projections

The planet Earth is a roughly spherical geoid. In order to represent it on flat computer screens, the surface of the Earth needs to be translated to a plane. This is realized by applying the method of a map projection which projects the bended, three-dimensional surface of the Earth onto a two-dimensional projection surface. The shape of the projection surface defines different possibilities of projection types as *planar*, *conical* and *cylindrical*. See figure **??** for a visual comparison of map projection types.

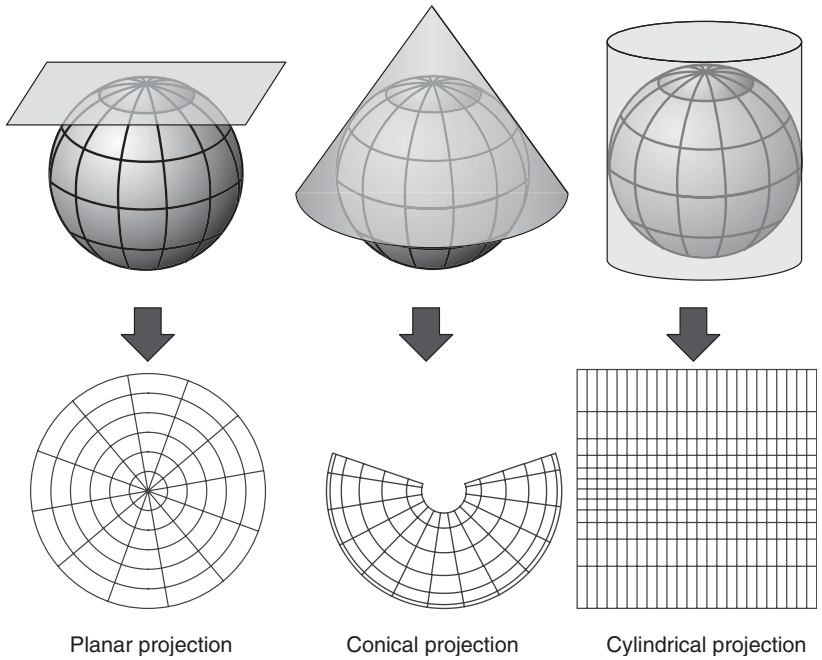


Figure 2.15: Types of map projections [**?**, p 28].

Flattening the curved surface of the Earth naturally causes *distortion* of different kinds, including areal, angular, scale, distance and direction distortion. Selecting a map projection influences which degree and combination of distortion will be caused. As no projection can optimize all those factors at once, choosing the right projection depends on the purpose of the map.

The **spherical mercator projection** is the most commonly used web mapping projection. Based on a normal cylindrical projection, it preserves local shapes and direction, but does this at the cost of enlarging areas towards the poles. As an example, Greenland appears on a Mercator map larger than South America while its actual size is $1/8$. The effect of this distortion can be visualized by Tissot's indicatrix. Figure **??** shows

how circles of the same relative size get extrapolated towards the poles when using the mercator projection [**?**, **?**, **?**].

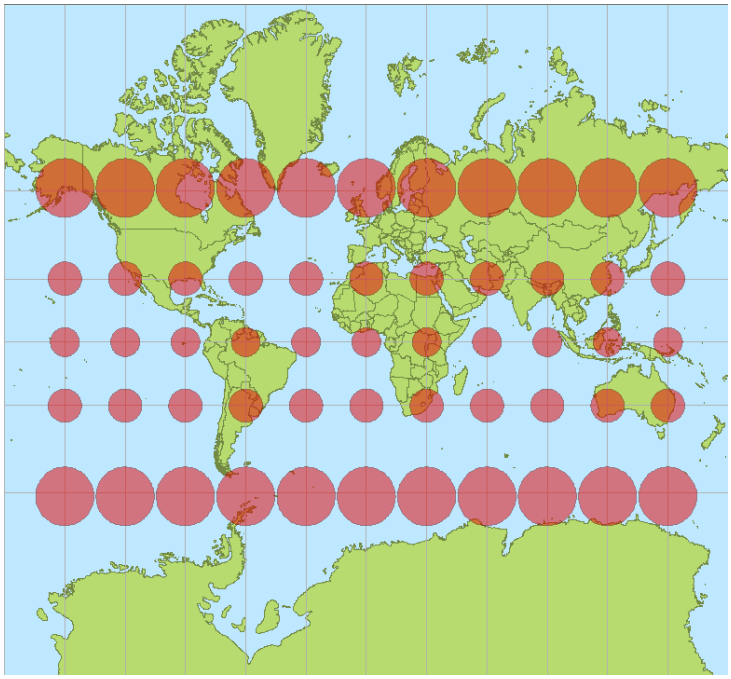


Figure 2.16: Tissot's indicatrix visualizes enlarged areas towards the poles when using the mercator projection [**?**].

Many countries have developed their own coordinate systems, such as the British National Grid or the German Gauß-Krüger coordinate system. They aim at reproducing the geographic regions within their territory in an appropriate manner. Standardization efforts go towards using the *Universal Transverse Mercator* projection. It avoids large distortions by comprising a series of Transversal Mercator projections that create separate grid zones with their own projections. This has the benefit of universally representing areas in a more exact way. On the other hand, coordinates need to be referenced including the zones in which they are located in [**?**].

2.4.3 Spatial data types

Two main representation types for spatial objects such as buildings, roads and other geometries exist: *vector data* and *raster data*. This mainly applies to 2-dimensional representation of spatial data, which often suffices the task for creating web maps and is preferred over 3-dimensional data handling in many cases for computational simplicity.

Figure **??** illustrates the conceptual different between both data types which will be discussed as follows.

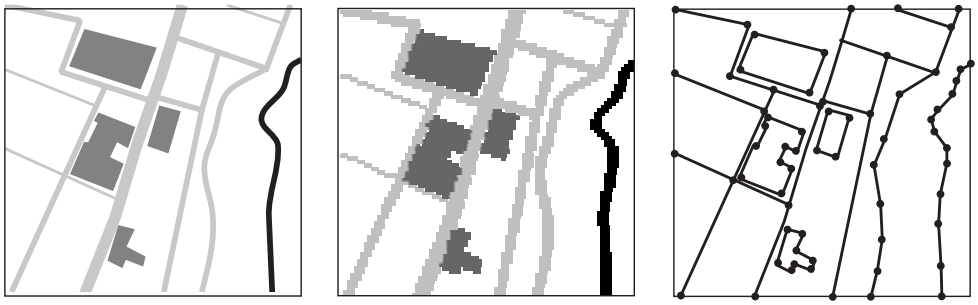


Figure 2.17: A map (a) displayed either in raster mode (b) or in vector mode (c) [**?**, page 38].

- **Vector data** is used to describe geometric shapes in a numeric way. *Points*, *lines* or *polygons* are specified by coordinates in a reference system.

  Simple points can be easily expressed as pairs of latitude, longitude values and stored within two separate columns within a database. More complex shapes like polygons require different data storage types, such as *Well Known Text* (WKT) or *Keyhole Markup Language* (KML).

- **Raster data** represents and stores geospatial data as a grid of pixels that forms a continuos surface. It is most commonly used for satellite imaginary. The arrangement of adjacent pixels intrinsically defines the spatial location of shapes within the raster image in relation to an externally defined reference system.

  The pixel values of the raster image usually depict a visual representation of the contained area, but they can also be assigned a specific meaning: The *digital elevation model* (DEM) for example is used to describe the average elevation of the mapped area on a per-pixel-basis. Raster images are also often generated from vector data by a tile renderer to create base layers for maps [**?**, **?**].

2.5 Visualization

In the following chapter, foundations for visualizing clusters on a map will be discussed. Basics of geographic visualization and their driving forces lead to abstraction and clus-

tering as a tool for simplifying information on maps. Visual variables, as well as classification approaches for geovisualization will be reviewed in order to discuss existing concepts for displaying aggregated data on a map.

Visualization is driven by the basic belief that 'seeing' is a good way of understanding and generating knowledge. Humans have a very well developed sense of sight, a fact which is underlined by more of 50 percent of the neurons in our brain being used for vision. [**?**].

MacEachren & Kraak [**?**] define that "Geovisualization integrates approaches from visualization in scientific computing (ViSC), cartography, image analysis, information visualization, exploratory data analysis (EDA), and geographic information systems (GISystems) to provide theory, methods, and tools for visual exploration, analysis, synthesis, and presentation of geospatial data (any data having geospatial referencing)." In his lecture notes on "Geographic visualization", Martin Nöllenburg adds that more human-centered definitions exist and observes that the user's needs have to be taken into account for effective geovisualization techniques [**?**].

The **goals of geovisualization** can be summarized using the *map use cube* by MacEachren and Kraak [**?**], which is illustrated in figure **??**. The goals *exploration, analysis, synthesis and presentation* are classified amongst three dimensions:

- The type of *task* varies from *knowledge construction* to *information sharing*. While the first is about revealing unknowns and constructing new knowledge, the latter will primarily share existing knowledge.
- The amount of *interaction* ranges from *high* to *low*. A low level of interaction means a rather passive consumption of knowledge, instead a high level will allow the user to actively influence the visualization.
- The *users* of visualization are classified between *public* and *private* audiences. A single, private user with specialized skills might require different visualization techniques than large, public audiences.

In the given model of the map use cube, the goals of exploring, analyzing, synthesizing and presenting shift between the extremes of the three defined aspects. The first goal of exploring is classified as a task of knowledge construction, based on high interaction and targeted at a rather private audience. On the other hand, presenting is a task of information sharing that requires a low amount of interaction but is suitable for public audiences [**?**].

Cognitive aspects of visualization help us understand, how visual thinking works. Complex input is abstracted on the retina of the human eye and matched against a vast collection of patterns from experience. Despite generating realistic images, visualization

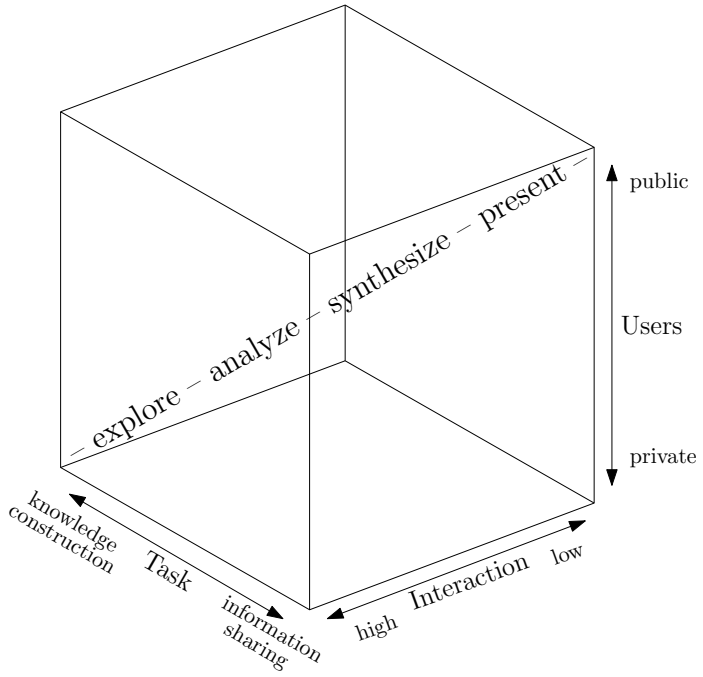


Figure 2.18: The map use cube after MacEachren and Kraak [**?**] characterizing geovisualization goals in a three-dimensional space by their level of interaction, their audience, and the addressed tasks. [**?**].

can help generate new ideas by using abstraction to communicate patterns. The idea is to allow the user to join insight, draw conclusions and interact with the data by presenting it in a visual form that reduces the cognitive work needed to perform a given task [**?**, **?**, **?**].

Various scientific publications [**?**, **?**, **?**, **?**, **?**, **?**, **?**] that have been researched for this thesis mention the importance of using abstraction for efficiently visualizing information. Especially maps can only highlight interesting information by filtering out unnecessary details of the environment. For example, a road map is better visualized on a clear background instead of satellite images that would distract the user from the primary goal of finding directions. The challenge is to balance realism and abstraction in geovisualization depending on the problem [**?**].

2.5.1 Visual variables

Information on a map is represented by symbols, point, lines or areas with properties such as color and shape. Bertin [**?**, **?**] has described the fundamental graphic variables

for map and graphic design. While being written for hand-drawn maps on paper, the concepts described by Bertin are still applied in todays digital mapping applications and have been further developed by MacEachren [**?**].

The main variables, introduced by Bertin are: *location*, *size*, *value*, *texture*, *color*, *orientation* and *shape*. MacEachren adds *crispness*, *resolution*, *transparency* and *arrangement* to the list and splits up color into its values of *brightness*, *saturation* and *hue*. Figure **??** describes a similar list of aesthetic attributes by different geometries and groups the variables regarding *form*, *color*, *texture* and *optics* [**?**].

Each variable has a different potential for visualizing data of categorical information (nominal and ordinal) or numerical information (including intervals and ratios). For example, the size of a point can be used to describe a numeric ratio and different shapes may be used to distinguish items based on categories. On the other hand, different texture patterns only offer a limited set of possibilities and size shouldn't be used for describing nominal properties [**?**, **?**].

2.5.2 Visual data exploration techniques

Depending on the task and the type of data to be shown, different forms of visualization and techniques for exploring the data exist. Daniel A. Keim [**?**] classifies such techniques using three criteria as depicted in figure **??**: the *data type* to be visualized, the *technique* itself and the *interaction and distortion* method [**?**]:

- The **data type** is classified into *one-dimensional* (as for example temporal data), *two-dimensional* (geographical maps), *multi-dimensional* (relational tables), *text and hypertext* (articles and web documents), *hierarchies and graphs* (networks) as well as *algorithms and software* (such as debugging operations).
- The **visualization techniques** are classified into *standard 2d/3d displays* (x-y plots and landscapes), *geometrically-transformed displays* (parallel coordinates), *iconic displays* (glyphs), *dense pixel displays* (recursive pattern) and *stacked displays* (tree maps).
- Finally, the third dimension describes the **interaction technique** being used, such as *dynamic projection*, *interactive filtering*, *zooming*, *distortion* and finally *link & brush* approaches.

With regards to visualizing clusters on a map, it appears that the visualization technique may be considered from two different view points:

- the visualization of the *entire map* with clustered points on it, as well as

- the visualization of an *individual cluster*, placed on the map.

A typical map for representing spatially clustered data is based on a least *two-dimensional data*, containing latitude and longitude information of cluster items. The map is presented in planar space, which classifies it amongst *standard 2d/3d displays*. Common interaction techniques for maps are *zooming* and *panning* which allow the user to explore the 2-dimensional space and reveal details.

The visualization of individual clusters on the map is likely to be classified amongst *iconic displays*. During the clustering process, individual points get aggregated, which potentially leads to multivariate aggregate information of item properties. Depending on the level of detail that clusters should expose, more complex visualization techniques may be possible.

Approaches for visualization techniques of the entire map and individual clusters will be discussed further in chapter **??**.

2.5.3 Clutter reduction

Clutter reduction is a way to enhance readability and general performance of maps. An early publication about *visual clutter* on maps by Richard J. Phillips and Liza Noyes [**?**] states that "reducing visual clutter improves map reading performance". Clutter reduces the background visibility and prevents the user from understanding structure and content of the data being presented. This especially becomes true for visualizing large data sets on maps, so that properties of the data items are hardly visible [**?**, **?**].

Clustering of course is the approach for clutter reduction that is primarily being investigated for this thesis. In order to review the effectiveness and limitations of clustering, as well as the relationship that it has to other techniques in that field, it is helpful to review the "Clutter Reduction Taxonomy" by Geoffrey Ellis and Alan Dix [**?**]. It distinguishes between three main types of clutter reduction techniques:

- **appearance**: alter the look of data items by using techniques like *sampling*, *filtering*, *changing point sizes*, *changing opacity* or *clustering*.
- **spatial distortion**: displace the data items in ways as *point/line displacement*, *topological distortion*, *space-filling*, *pixel-plotting* or *dimensional reordering*.
- **temporal**: use *animation* to reveal additional information.

In a next step, the stated techniques have been evaluated against a list of 8 high-level criteria [**?**]. For this thesis, the relevant information for the clustering technique have been extracted and will be outlined by each criterium:

1. *avoids overlap*
   The major benefit is to reduce clutter, provide the ability of seeing and identifying patterns, have less hidden data, as well as giving more display space to points. Clustering can be used in such a way to avoid *overplotting* by representing groups of points as single points.

2. *keeps spatial information*
   Maintaining the correct coordinates of items is relevant, but the study also states that relative positions of points might have greater influence on orientation than just their exact coordinates. Clustering by definition looses the spatial information of individual points, but using aggregate values, like the centroid a cluster, as the position can be used for compensation.

3. *can be localized*
   The term localization is used to specify, if the display can be reduced to a specific region. This is usually provided by *focus and context* techniques that reveal information underneath by zooming into an area. The study doesn't make a clear decision regarding the applicability of localization for clustering and states that different properties for spatial and non-spatial clustering apply. In the case of spatial clustering, localization is definitely possible and has been implemented, see chapter **??**.

4. *is scalable*
   Scalability of the clutter reduction technique with regards to large amounts of data is the goal of this criterium. The study admits that the meaning of large datasets is vaguely quantified. As one of the main goals for this thesis is to enhance performance for large data sets by using clustering, the technique is expected to satisfy this condition. Of course, the range of scalability depends on the implemented clustering algorithm. Also refer to the time complexity definitions in chapter **??**.

5. *is adjustable*
   If the user is able to control aspects of the visual display and adjust parameters of the system that influence the degree of display clutter. Scientific methods in cluster analysis tend to offer a a higher level of interactivity, while public facing clustering applications limit the amount of interactivity to controlling cluster sizes and the previously discussed localizing feature.

6. *can show point/line attribute*
   The goal is to map attributes of the data to properties like color, shape or opacity of the displayed points or lines. For clustering, this feature can be used in order to display aggregates of the multivariate data results from the clustering process. Further information can be found in chapter **??**.

7. *can discriminate points/lines*
   Being to able to identify individual data items within a crowded display is the goal of this criterium. The study states the capability of clustering for detecting outliers as well as creating groups of points in order to satisfy this criterion. On the other hand, this classification appears unclear, as the grouping process of clustering aggregates individual information by definition and only makes it accessible by request or localization.

8. *can see overlap density*
   This helps to gauge the amount of overplotting, see where higher density regions are and understand the distribution of data underlying the visualization. Clustering can show the amount of items within clusters by using visual indicators as point size, opacity and color.

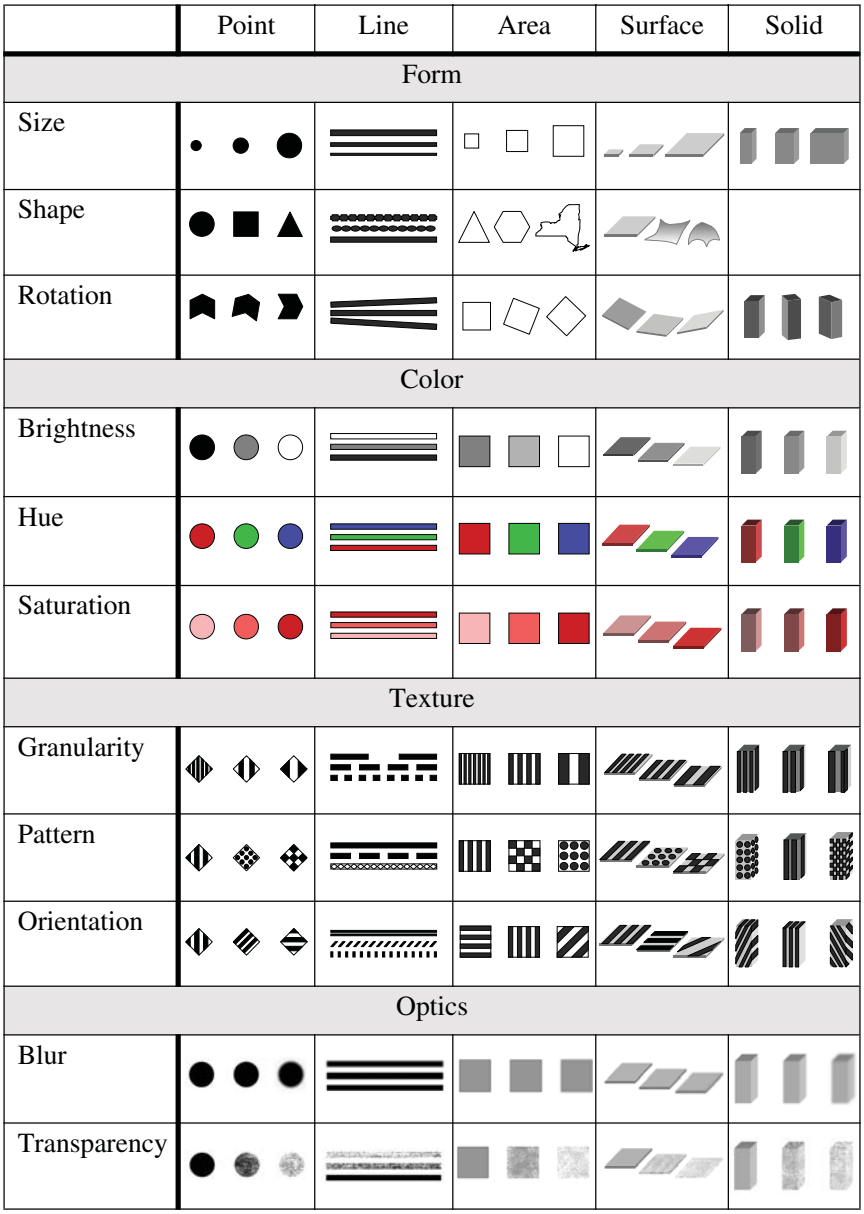


Figure 2.19: Aesthetic Attributes by Geometry [**?**].

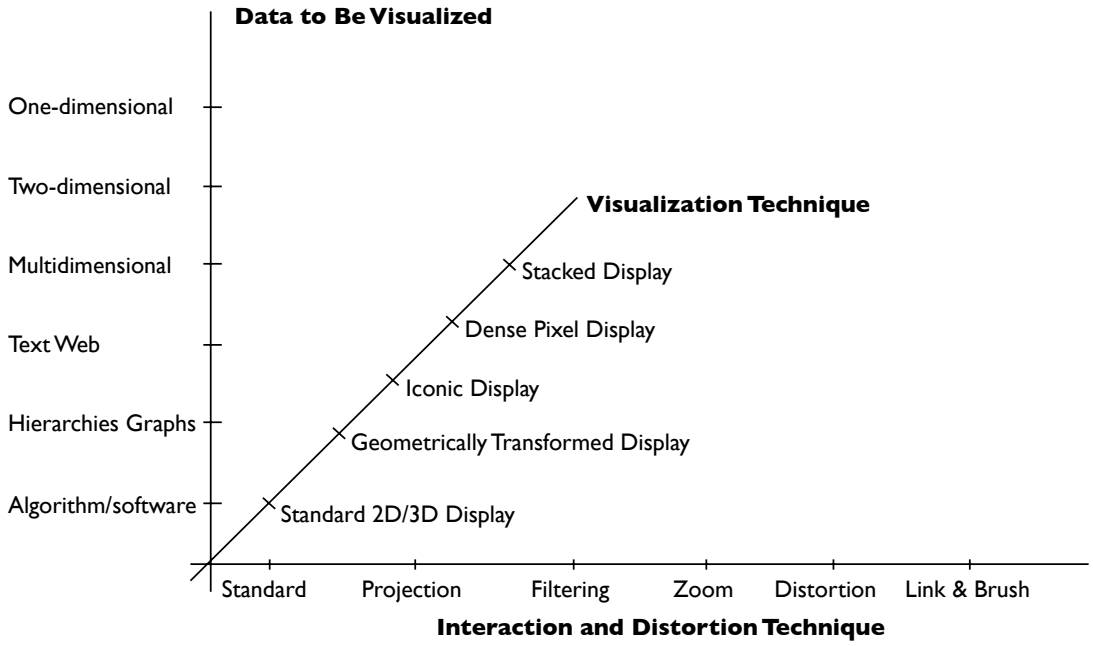


Figure 2.20: Classification of visual data exploration techniques based on [**?**].

CHAPTER 3

State of the Art

The following chapter discusses the state of the art of related technologies for the thesis. A modern web mapping stack is explained as well as the basics of Drupal & mapping technologies. Further, the state of the art for client-side and server-side clustering technologies in web mapping will be analyzed.

3.1 A Modern Web Mapping Stack

The primary purpose of web mapping applications is to deliver spatial data in the form of a map to the user. Modern, interactive web mapping applications are based on a concept named *slippy maps*. These maps are brought to the user by a combination of client- and server-side technologies. Figure **??** illustrates the prototype of such a modern web mapping application. Similar mapping stacks are documented in [**?**, **?**, **?**, **?**]. Its components will be discussed in the following section.

- A **slippy map** is displayed to the user in a rectangular viewport within the browser and handled by a JavaScript mapping library. The map is visualized dynamically by rendering layers of raster and vector data on top of each other. In addition, the slippy map provides means of interaction to the user such as panning and zooming to update and explore the map.
- A **JavaScript mapping library** is in charge of rendering the slippy map by positioning one or multiple layers on top of each other. Usually, a base layer of raster

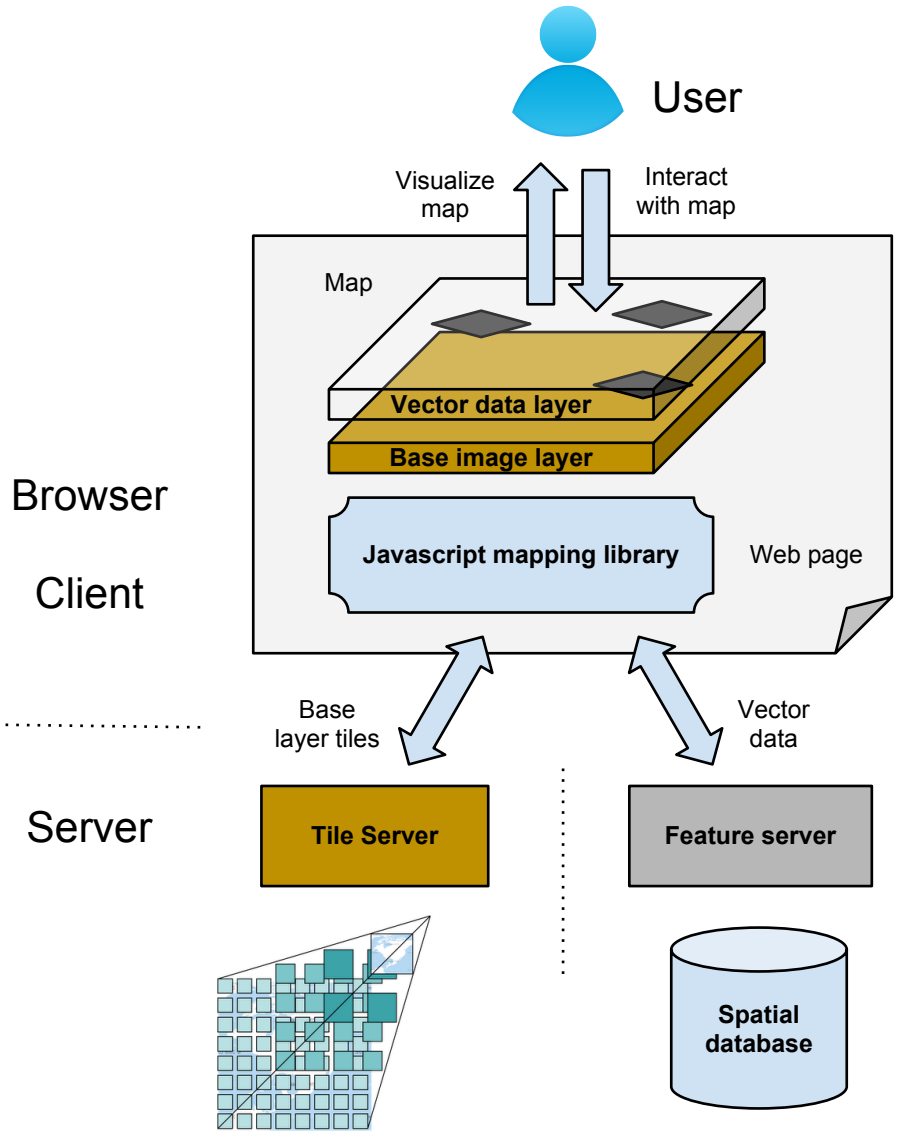


Figure 3.1: Illustration of a modern web mapping application. Includes a tile graphic from [**?**].

data is combined with a vector data layer on top of it. Current JavaScript mapping libraries include *OpenLayers*[1], *Leaflet*[2] and *Modest Maps*[3].

[1] http://openlayers.org/
[2] http://leafletjs.com/
[3] http://modestmaps.com/

- A **tile server** provides static, sliced-up images of the raster image data for creating base layers of the slippy map. The JavaScript mapping library requests on these tiles on demand, based on the current viewport. When the user performs actions like dragging and zooming on the map, the current viewport will get changed. The JavaScript library then requests additional data from the server, if needed and updates the map accordingly. This live-updating process is also referred to as *Bounding Box Strategy* (BBOX Strategy)[4]. Standards for consuming tiles include *Web Map Service* (WMS)[5] and *Tile Map Service* (TMS)[6].

  A typical tile set represents 256×256 pixel tiles of the whole world at 18 zoom levels. The fact that this leads to billions of tiles has created its own line of businesses. Map providers like *Google Maps*[7] and *Stamen*[8] both offer default base layers and also allow users to create their own custom tile sets. *TileMill*[9] is an open source project that allows users to design their own custom maps using a design studio. The resulting maps can either be self-hosted on a server by using its complement *TileStream*[10] or imported into *MapBox Hosting*[11].

- A **feature server** provides vector data to the JavaScript mapping library in an analogous way as the tile server does. Note the difference: while raster data will be displayed directly as an image, the vector data gets rendered on the client-side.

  To provide dynamic data, the feature server usually relies on a *spatial database*. Spatial extensions exist for various databases, including *PostGis*[12] and *MySQL Spatial Extensions*[13] and *Spatialite*[14].

3.2 Drupal & Mapping

The book "Mapping with Drupal" [**?**] by Alan Palazzolo and Thomas Turnbull was published at the end of 2011 and gives a throughout overview of available mapping modules for Drupal 7 by that time. As the web, Drupal is a constantly evolving plat-

[4] http://openlayers.org/dev/examples/strategy-bbox.html
[5] http://en.wikipedia.org/wiki/Web_Map_Service
[6] http://en.wikipedia.org/wiki/Tile_Map_Service
[7] http://maps.google.com/
[8] http://maps.stamen.com/
[9] http://mapbox.com/tilemill/
[10] https://github.com/mapbox/tilestream
[11] http://mapbox.com/hosting
[12] http://en.wikipedia.org/wiki/PostGIS
[13] http://dev.mysql.com/doc/refman/5.0/en/spatial-extensions.html
[14] http://www.gaia-gis.it/gaia-sins/

form. Development continues, so its a permanent exercise to keep up-to-date with recent technologies.

3.2.1 Drupal

Drupal is a free and open source content management system and framework[15]. Developed and maintained by an international community, it currently backs more than 2% of all websites worldwide. It is used to create personal blogs, corporate and government sites and is also used as knowledge management and business collaboration tool[**?**].

Drupal is a set of PHP scripts that can be extended by using existing and creating custom modules. It is beyond the scope of this thesis to explain Drupal in detail, but the following terms will be used throughout this thesis:

- **hooks**: while using APIs and object-oriented programming are familiar concepts, *hooks* are a Drupal-specific concept. They basically allow modules to interact with each other in a procedural way in which many parts of the Drupal system are built.

  > Drupal's module system is based on the concept of "hooks". A hook is a PHP function that is named foo_bar(), where "foo" is the name of the module (whose filename is thus foo.module) and "bar" is the name of the hook. Each hook has a defined set of parameters and a specified result type.
  >
  > To extend Drupal, a module need simply implement a hook. When Drupal wishes to allow intervention from modules, it determines which modules implement a hook and calls that hook in all enabled modules that implement it.[16]

- **Entities and Fields**: Since Drupal 7, the generic concept of *entities* provides content and data containers which are used in Drupal to represent content like *nodes* but also *users*, *comments*, etc. *Fields* empower site builders to assign fields to various entity types which allows for a flexible content modeling framework[**?**].

- **patches**: the term *patch* describes a way to submit code changes used within the Drupal and other open source communities:

[15] http://drupal.org
[16] http://api.drupal.org/api/drupal/includes!module.inc/group/hooks/7

> Patches are pieces of code that solve an existing issue. In fact, patches describe the changes between a before and after state of either a module or core. By applying the patch the issue should no longer exist.
>
> Patches are used to maintain control-ability over the entire Drupal project. While Drupal is distributed via the git version control system, patches are additional pieces of code that focus on a single change request and therefore are easily tested, reviewed and documented.[17]

In the following, mapping-related technologies for Drupal 7 are explained.

3.2.2 Data storage

The **Geofield**[18] module is stated to be the "best choice for all but the simplest mapping applications" [**?**, page 27] in Drupal 7. It is based on the concept of *Entities* and *Fields* that has been introduced in the previous section. By using the *geoPHP library*[19] for geometry operations, it allows to store location data in various formats as latitude/longitude, WKT. Geofield integrates with various other mapping modules related to data input and storage in Drupal, as illustrated in figure **??**.

Besides the Geofield module, the following modules are relevant when considering ways to store location data in Drupal:

- **PostGIS integration** is often requested when dealing with complex spatial data. A variety of modules and approaches exist for integration with PostGIS. The *PostGIS* module[20] is similar to Geofield, but relies on PostGIS for spatial operations and data storage. *Sync PostGIS*[21] allows to push geospatial data from a Drupal-internal storage as Geofield into PostGIS. Recently, a pluggable storage backend was added[22] to the Geofield module in order to allow integration with alternative databases. *Geofield PostGIS*[23] therefore is a more integrated option for storing Geofield data in PostGIS.
- **Solr integration** is another common approach when creating maps based on Drupal. *Apache Solr search* is a fast open source search platform written in the Java programming language. There are two common modules for Solr integration in

[17] http://drupal.org/patch
[18] http://drupal.org/project/geofield
[19] https://github.com/phayes/geoPHP
[20] http://drupal.org/project/postgis
[21] http://drupal.org/project/sync_postgis
[22] http://drupal.org/node/1728530
[23] https://github.com/phayes/geofield_postgis

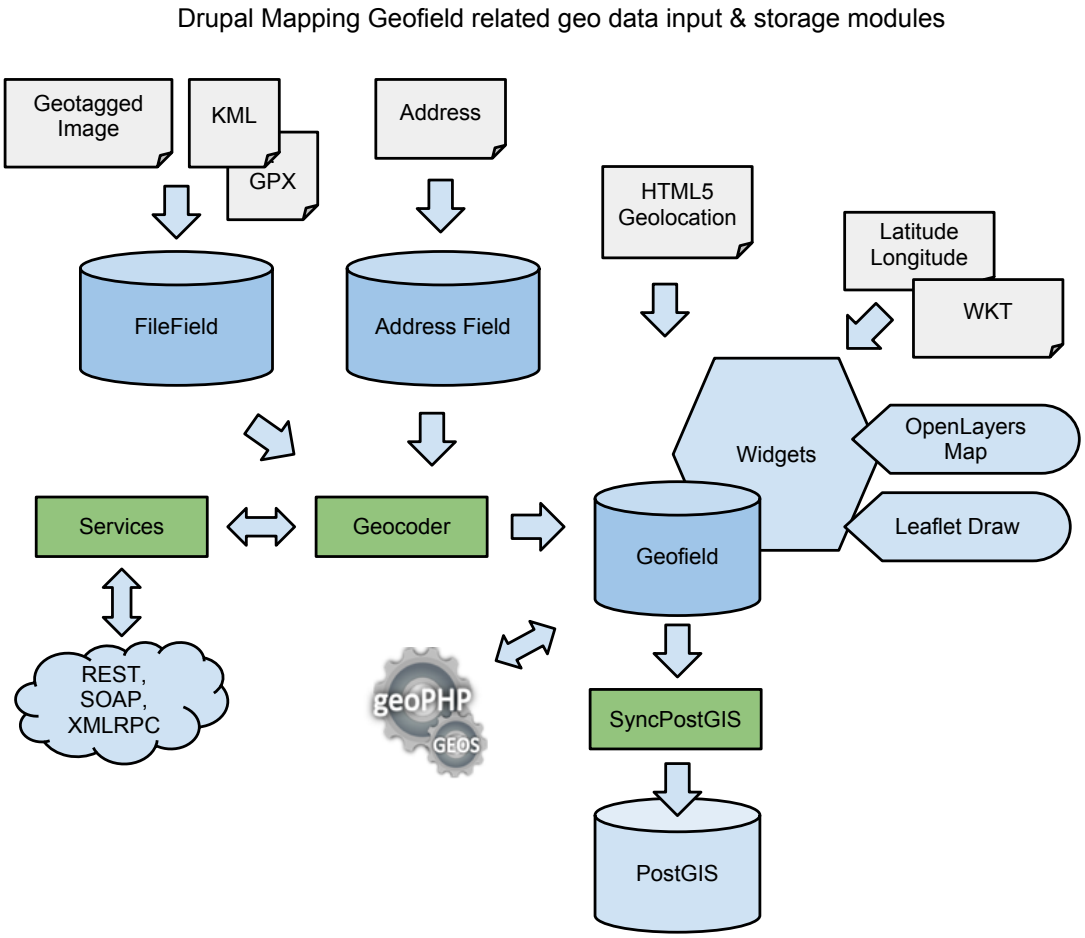


Figure 3.2: The Drupal Geofield module and related geo data input and storage modules.

Drupal, which both offer support for indexing spatial data: *Search API*[24] and *Apache Solr Search Integration*[25].

- The **Location**[26] module is another popular choice for storing spatial data in Drupal 7. As its architecture doesn't follow current Drupal conventions, its rather a monolithic system that provides a rich out-of-the-box experience but doesn't integrate that well with other modules like Geofield does [**?**].

[24] http://drupal.org/project/search_api
[25] http://drupal.org/project/apachesolr
[26] http://drupal.org/project/location

3.2.3 Data presentation

Being a content management system and framework, the second most important task for handling spatial data in Drupal is presenting it in various ways. Again, a variety of modules exists for querying and displaying geospatial data. Figure **??** illustrates how the query- and display-related Drupal mapping modules work together in a common use case.

A **mapping module** provides the basic integration for a JavaScript mapping library with the Drupal internals. The most widely used mapping modules are the *OpenLayers* module[27] with 8,325 active installations on Drupal 7 by April 2013 and the GMap module[28] with 24,113. The Leaflet module[29] only counts 838 active installations reported on drupal.org by that time, but has the advantage of being more lightweight than OpenLayers and is not bound to a single, commercial API such as GMap.

The interaction with the spatial data provided by the Geofield module can be classified into three different scenario types, visualized by circles within figure **??**:

1. In the first scenario, the mapping module directly accesses data from Geofield.

   This approach is usually applied when displaying maps for single single pieces of content with location. The geo data retrieved from Geofield is then transferred to the client within the HTML response. On the client-side, the JavaScript mapping library takes care of visualizing the geo data.

2. In the second scenario, integration with the **Views** module is used to query a collection of data.

   The Drupal *Views* module[30] is the de-facto standard for creating listings of content of any kind. With 422,100 reported installations in Drupal 7 by April 2013, it is the most widely used module and it also will be part of Drupal 8 core. This powerful tool allows site builders to configure database queries using an administration interface. In addition, the module provides formatting options for representing query results. In combination with extension modules, Views allows to create lists, tables and many other formats based on a collection of dynamic data. Using Views allows the mapping module to query a listing of locations from Geofield, based on user-defined parameters. Instead of returning single locations as in scenario one, the second therefore processes a collection of geo data values.

[27] http://drupal.org/project/openlayers
[28] http://drupal.org/project/gmap
[29] http://drupal.org/project/leaflet
[30] http://drupal.org/project/views

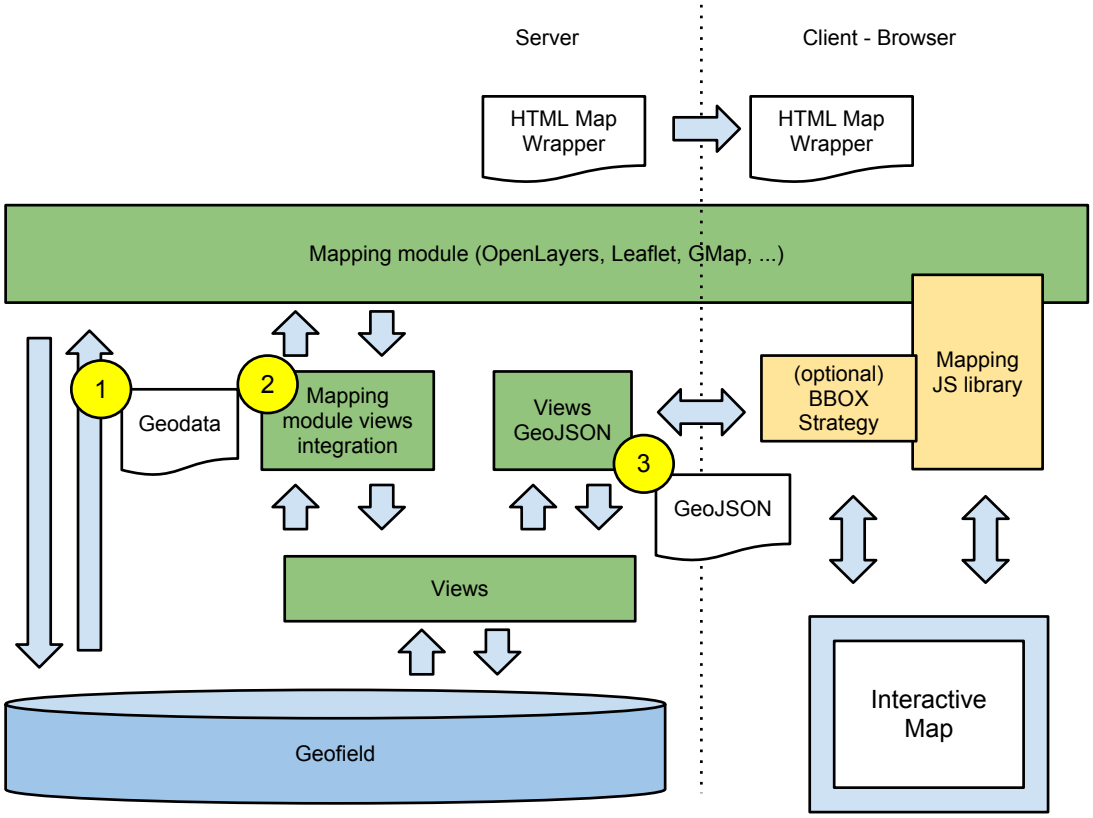


Figure 3.3: The prototypic work-flow of query and display-related Drupal mapping modules.

3. Scenario three allows for dynamically updating the map based on user interaction.

   In this case, the geospatial data is not delivered within the HTML response, as in the previous approaches. The JavaScript library issues a separate request to the server using a **Bounding Box strategy**. The OpenLayers JavaScript library contains such a BBOX Strategy[31] and the *Leaflet GeoJSON* module[32] provides the same for the Leaflet library. The strategy essentially requests the geo data within the bounding box of the current viewport. On the server-side, the *Views GeoJSON* module[33] initiates the query and transforms the data returned by Views into a *GeoJSON*[34] feed in order to deliver it to the JavaScript mapping library on

[31] http://dev.openlayers.org/docs/files/OpenLayers/Strategy/BBOX-js.html
[32] http://drupal.org/project/leaflet_geojson
[33] http://drupal.org/project/views_geojson
[34] http://www.geojson.org/

the client-side.

Note, that the exact implementation details vary between the used mapping modules.

Solr integration for querying and displaying geospatial data in Drupal is mainly provided by the integration of the Solr-related modules with Views. For the *Apache Solr Search Integration* module exists a *apachesolr_location* module[35] and a *ApacheSolr Geo* sandbox project[36]. For *Search API* there is a *Search API Location*[37] module.

3.3 Clustering implementations in Web Mapping

Clustering in web mapping affects the way how vector data is processed and represented to the user. According to the web mapping stack described in **??**, we can differentiate between client-side and server-side clustering. A server-side clustering implementation will cluster the data already before sending it over the network to the browser client. In a client-side clustering implementation, the client receives the unclustered data set from the server and clusters it on its own.

Client-side vs. server-side clustering

Client-side clustering is convenient because of several reasons. The clustering task can be abstracted from the server without the need to account for server-side implementation details. It also relieves the server from performing the clustering task which can positively influence scalability. Executing the clustering task on the client-side allows for better interaction: the user may zoom into or expand clusters without the need for an additional request to the server.

On the other hand, client-side clustering forces the server to deliver the entire data set. This also means that a bigger amount of data has to be processed and transferred over the network. Subsequently, the client needs to cope with receiving the larger data set and takes over the burden of clustering the data.

3.3.1 Client-side clustering in Web Mapping

When clustering on the client-side, the JavaScript mapping library receives the entire, unclustered data set and executes a clustering algorithm before visualizing the data to the user.

[35] http://drupal.org/project/apachesolr_location
[36] http://drupal.org/sandbox/pwolanin/1497066
[37] http://drupal.org/project/search_api_location

- The **Leaflet.markercluster**[38] library provides animated marker clustering for the Leaflet JavaScript mapping library. It combines the *agglomerative hierarchical clustering algorithm* with a *distance grid* (see chapters **??** and **??**). The library features advanced cluster visualization techniques for representing shapes of clustered items and animations. When the user zooms into the map, clusters get expanded in a visual way and they collapse in the opposite direction.

  Leaflet.markercluster leverages the advantages of being a client-side implementation by implementing a *hierarchical clustering* approach that precalculates the clusters for all zoom levels. The markers are inserted into a distance grid on the lowest zoom level. The grid is then used to check for overlapping neighbors. If the inserted marker needs to get merged, this information is automatically propagated to upper levels within the hierarchy. Otherwise, the same checking procedure will be repeated for the inserted marker on the next, upper level.

  The cluster visualization of Leaflet.markercluster is supported by the *QuickHull*[39] algorithm to compute the enclosing *convex hull* of the clustered points as illustrated in b) of figure **??**. In addition, a *spiderfier* algorithm allows the user to select clustered points, even if they are positioned very closely to each other, see a) in figure **??**.

  The clustering task is computed in *linear time* to the number of markers *n* and the usually constant number of zoom levels.

- The **OpenLayers Cluster Strategy**[40] is included in the OpenLayers library and provides a simple *distance-based*, client-side clustering.

  When creating the clustering, the features are sequentially inserted. Every new feature is compared against all existing clusters. If the new feature overlaps with any cluster, it will get merged into the existing cluster. Otherwise, the feature is inserted as its own cluster. Once the data, viewport or zoom level changes, the clustering process will be re-initiated.

  The sequential insertion and the comparison to all existing clusters leads to a *factorial time complexity* of the algorithm.

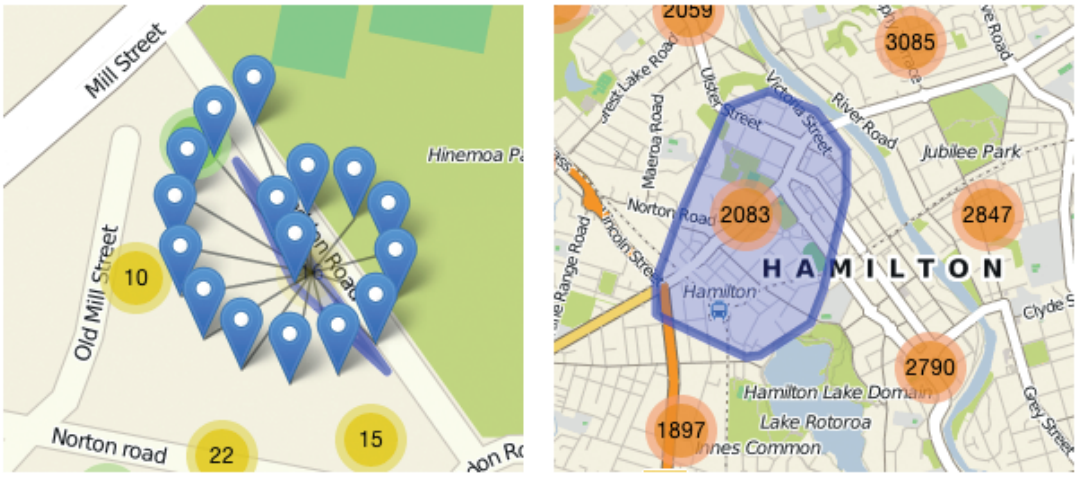
- **k-means clustering**[41] is a clustering library for the *Polymaps* JavaScript mapping library. It leverages the k-means squared error algorithm discussed in chapter **??** to create clusters in linear time. As discussed, the k-means algorithm computes in *linear time*.

[38] https://github.com/Leaflet/Leaflet.markercluster/

[39] http://en.wikipedia.org/wiki/QuickHull/

[40] http://dev.openlayers.org/releases/OpenLayers-2.12/lib/OpenLayers/Strategy/Cluster.js

[41] http://polymaps.org/ex/kmeans.js



a) b)

Figure 3.4: Two screenshots taken from the Leaflet.markercluster example map: a) spiderfied representation to select from multiple overlapping points and b) the visualized convex hull of a cluster indicates the real shape of the cluster on mouse-hover.

Other client-side clustering libraries evaluated can be classified similarly to the previously discussed ones. Grid-based approaches similar to Leaflet.markercluster include the Clustr library[42] for Modest Maps and Clusterer2[43] for Google Maps. MarkerClustererPlus for Google Maps V3[44] takes an approach similar to the OpenLayers Cluster Strategy.

3.3.2 Server-side clustering in Web Mapping

Server-side implementations cluster the data already before sending it over the network to the client.

While client-side clustering solutions are standardized towards available JavaScript mapping libraries, this doesn't apply to server-side implementations. Clustering on the server-side can be performed both on the database and/or application layer. The wide variety of tools involved in the server-side mapping stack from different spatial databases

[42] https://github.com/mapbox/clustr

[43] http://www.acme.com/javascript/#Clusterer

[44] http://code.google.com/p/google-maps-utility-library-v3/wiki/Libraries#MarkerClusterer

and programming languages seems to make it harder to establish off-the-shelf libraries for server-side clustering, as they always need to be integrated into a certain architecture.

- **Clustering maps** by Wannes Meert is the only scientific publication found that explores server-side clustering for web mapping. The study compares strengths and weaknesses of three different approaches (*K-means*, *Hierarichal clustering* and *DBSCAN*) and finally implements density-based clustering. The solution is based on the *DBSCAN* algorithm and uses the R-tree as indexing structure to speed up neighborhood searches. It is implemented and documented as a *PHP server-side script*. The time complexity of the implemented and discussed final algorithm is quasilinear: $O(n \log n)$ and the demonstration page[45] asserts a clustering time of ~ 1 second for only 525 points [**?**].

- **Google Maps Hacks: Tips and Tools for Geographic Searching and Remixing** is a book that contains a section *Hack 69. Cluster Markers at High Zoom Levels* that discussed various approaches to server-side cluster markers for Google Maps. The chapter evaluates different approaches from a quadratic-time $O(n^2)$ implementation of the *k-means* algorithm over a even worse cubic-time $O(n^3)$, *hierarchical clustering* approach. Finally, a *naive grid-based clustering* solution is developed and documented as a *CGI Perl Script* which is stated to perform in linear time $O(n)$ [**?**].

- **Geoclustering**[46] is a Drupal 6 module designed to "scale to 100 000+ markers". By maintaining a cluster tree in a separate database table, it essentially pre-calculates clusters. Similar to the grid-based clustering algorithm *STING* described in **??**, by shifting the clustering complexity to the calculation of the cluster tree, queries can be performed in constant time, only linear to the number of grids $O(g)$. This speed improvement reduces the possibilities of dynamic filtering to bounding-box filters only. Filters on other properties of the indexed data would need to be pre-calculated into a more complex or separate cluster trees.

- **Clustering in Apache Solr** has been researched as well. While Solr integrates Carrot2 as a document clustering engine[47], spatial clustering isn't supported out-of-the-box. A tutorial written in German documents the conceptual implementation of grid-based clustering in Solr[48]. As Geohash support has been added to Solr to support proximity searches [**?**], it could be used as well for spatially clustering data.

[45] http://www.wannesm.be/maps/
[46] https://github.com/ahtih/Geoclustering
[47] http://searchhub.org/2009/09/28/solrs-new-clustering-capabilities/
[48] http://blog.sybit.de/2010/11/geografische-suche-mit-solr/

Other server-side clustering implementations have been researched:

On the **database-layer**, *SQLDM - implementing k-means clustering using SQL* describes a linear-time implementation of the k-means clustering algorithm in *MySQL*[**?**]. The *PostGIS* spatial extension for the *PostreSQL* database provides grid-based clustering via *ST_SnapToGrid*[49]. Also, a k-means module[50] (also see [51]) and documentation for clustering indices based on *Geohash* [52] for PostGIS exist.

Further, **application-layer** clustering implementations are grid- and distance-based clustering for Google Maps in PHP[53]. Vizmo is a research project that developed a server-side clustering component in PHP, Symfony2 and MySQL[54]. Beyond the primary scope of PHP applications, a Flex-based clustering implementation has been developed for ArcGIS server[55]. ClusterPy[56] is a library of spatially constrained clustering algorithms in Python and SGCS[57] is a package to build graph based clustering summaries for spatial point patterns in R.

3.4 Visual mapping

Nöllenburg [**?**] names three "Driving forces in geovisualization": The advent of high speed parallel processing and technology advances in computer graphics, today allows us to grasp enormous amounts of information. Besides the advances in graphics and display technologies, the second main driving force in geographic visualization is the increasing amount of geospatial data being collected and available. Finally, the third force is the *rise of the Internet*, which significantly pushes web mapping and contributes to geovisualization technologies.

As a logical consequence of broader audiences having access to geospatial visualizations using the Internet, it appears that there is a shift from technology-driven visualization towards more human-centered approaches. Interactive and highly dynamic interfaces have helped the map evolve from its traditional role as a presentational device towards exploring geospatial data [**?**, **?**].

[49] http://postgis.refractions.net/docs/ST_SnapToGrid.html
[50] http://pgxn.org/dist/kmeans/doc/kmeans.html
[51] http://gis.stackexchange.com/questions/11567/spatial-clustering-with-postgis
[52] http://workshops.opengeo.org/postgis-intro/clusterindex.html#clustering-on-geohash
[53] http://www.appelsiini.net/2008/11/introduction-to-marker-clustering-with-google-maps
[54] http://www.globalimpactstudy.org/wp-content/uploads/2011/12/vizmo-poster.pdf
[55] http://thunderheadxpler.blogspot.co.at/2008/12/clustering-20k-map-points.html
[56] http://www.rise-group.org/risem/clusterpy/index.html
[57] http://cran.r-project.org/web/packages/SGCS/

In chapter **??**, foundations of visualization, visual variables and data exploration techniques, as well as the concept of clutter reduction have been introduced. In the following, existing visualization techniques for representing clustered data on maps will be discussed. In a first step, visualization concepts on the map level will be investigated. Later, approaches for visualizing individual clusters are added. An evaluation summarizes the discussed technologies.

3.4.1 Map visualization types for clustering

There exists a variety of map types, serving different purposes like standard *geographic maps*, *cartograms*, *geologic maps*, *linguistic maps* or *weather maps*. Some of these use distortion, for example the cartogram can be used to map the area of each country to the size of population. In this section, we try to identify those map types which are appropriate for visualizing clustered data [**?**, **?**]:

- **Geographic map with markers**. The default way of representing data is a standard 2-dimensional map with markers on top of it. Each marker represents a data point, or in the case of clustering, a cluster.

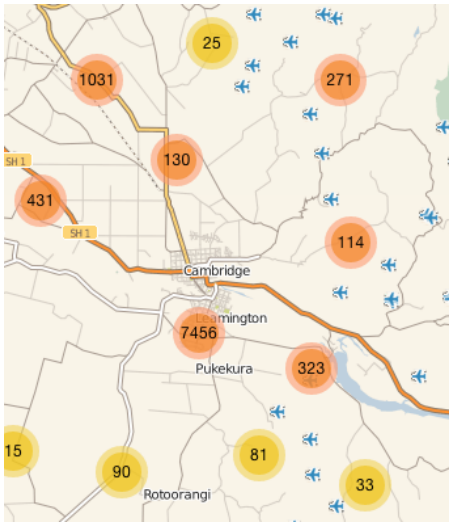


Figure 3.5: Leaflet map

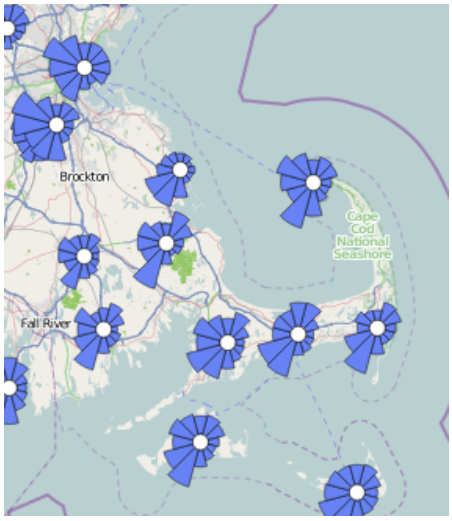


Figure 3.6: Wind history map

  - Figure **??** depicts an example[58] of the Leaflet.markercluster library. Clustered results are displayed using markers of the same size. The amount of

[58] Leaflet.clustermarker example map from http://leaflet.github.io/Leaflet.markercluster/example/marker-clustering-realworld.10000.html

items within a cluster is indicated using text within the markers and by using a "hot-to-cold" color ramp [**?**].

– Figure **??** shows a similar example, in this case a Wind history map[59] with markers for every wind measurement point. An advanced visualization technique is used for visualizing the amount of wind per cardinal direction as *polar area diagram*.

These two variations of standard maps shall indicate the potential of using advanced visualization techniques for displaying cluster items. Further ways for cluster visualization will be discussed in chapter **??**.

- **Geographic Heat map**

  Heat maps use colored, two-dimensional areas to express the value of each data entity on the map. *Choropleth maps* are the most common heat maps, which are often used for analysis of geographic and statistical data.

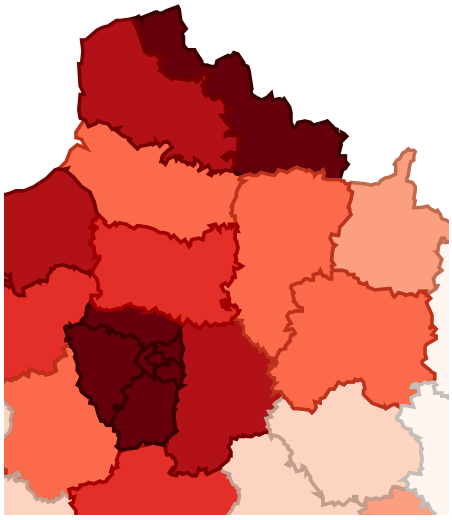


Figure 3.7: Choropleth map[60]

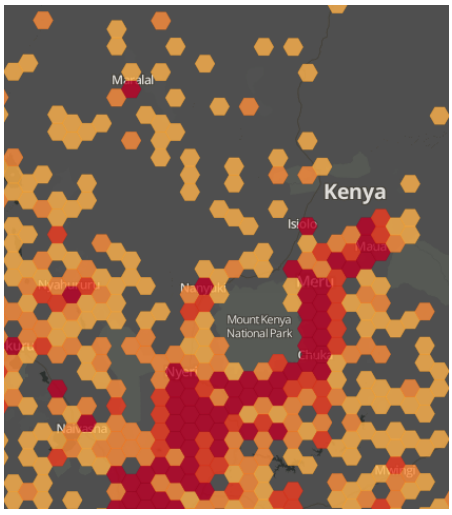


Figure 3.8: Heat map[61]

– Figure **??** visualizes an example choropleth map that shows population data for each of the departments of metropolean France. Color coding is used to indicate densely populated regions with heavier red tones.

– Figure **??** presents another example that uses *binning* for creating a hexagonal tessellation of the surface in order to visualize clustered results.

[59] Wind history map http://windhistory.com/map.html
[60] Choropleth map example from Kartograph http://kartograph.org/showcase/choropleth/
[61] Heat map that uses binning http://mapbox.com/blog/binning-alternative-point-maps/

A problem with heat maps is that they require a non-overlapping tessellation of the surface to provide the areas for visualization. As the binning example indicates, such a tessellation can be done programmatically. Heat maps therefore can also be used for visualizating arbitrary clustered data without a need to calculate the exact boundaries of clusters. Another variation of heat maps is the *prism map*, which adds extrusion of areas as a third dimension [**?**, **?**]. A publication on the evaluation of color schemes in choropleth maps can be found in [**?**].

- **Dot grid maps**

  Dot grid maps are based on the suggestion by Jaques Bertin [**?**, **?**] to use graduated sizes in a regular pattern as an alternative to chloropeth maps. The advantage is that the map creator doesn't have to choose between quantity or density of a distribution value, because the dot grid map shows both a the same time. The user can understand the data distribution on a finer level of granularity, as opposed to where the chloropeth map usually creates larger areas of aggregated information [**?**].

  Figure **??** is an alternative version of the France map from figure **??**, visualized as a dot grid map.

- **Voronoi map**

  The Voronoi tessellation is a space partitioning technique. From a set of points it produces a Voronoi polygon for each point, such that the area covered is closest to that point in comparison to all other points. Jean-Yves Delort describes a technique that uses Voronoi polygons for "Vizualizing Large Spatial Datasets in Interactive Maps" [**?**]. It uses a hierarchical clustering technique to choose a subset of points per zoom level for proper visualization. Still, the effectiveness of this approach is questionable, as the scalability analysis of the studies shows that the technique can efficiently be used for datasets of up to 1000 items.

  Figure **??** shows an exemplary voronoi map that displays all U.S. airports as of 2008. Besides the shown visualization, for Voronoi maps apply the same visualization possibilities as for cloropeth maps.

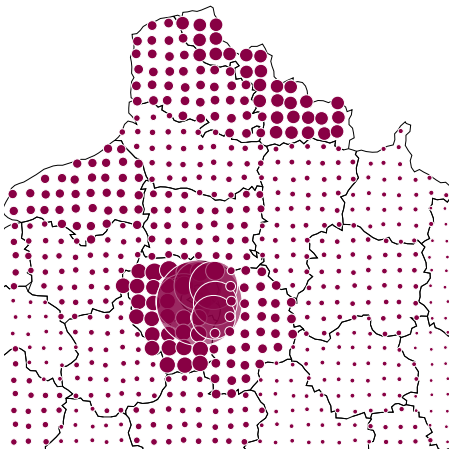


Figure 3.9: Dot Grid map[62]

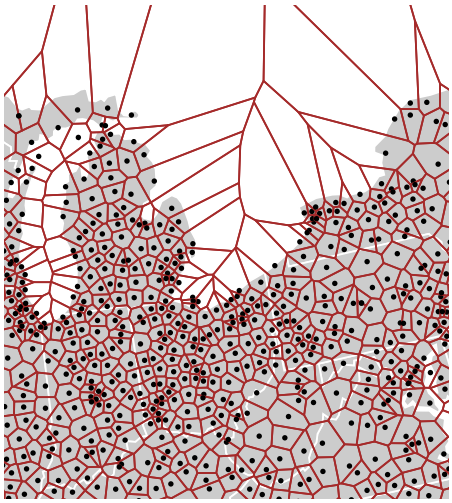


Figure 3.10: Voronoi map[63]

Self-organizing maps (SOM) can also be used to visualize clusters of data. But instead of displaying data on a geographic map, self-organizing maps create their own virtual space in order to represent information [**?**].

3.4.2 Cluster visualization techniques for maps

The previous chapter has shown different kinds of map visualization techniques appropriate for displaying clustered data. In the end, each visualization will show (clustered) items on a map as visual objects with attributes like a particular shape or coloring. As clusters contain aggregated information, an important task is to find the right way for visualizing the cluster items themselves. From the examples provided so far, we have seen variations in size, shape and color which expose information on the cluster items being visualized on the map. In the following, multivariate data visualization techniques will be evaluated for visualizing cluster items on a map.

In chapter **??**, a classification of visualization techniques by data type and interaction technique has been presented. Ke-Bing Zhang [**?**] has written about "Visual Cluster Analysis in Data Mining", where he list an extensive list of multivariate data visualization techniques. Potentially, any such visualization technique can be used, but the frame of the map puts constraints in terms of space on the representation of individual items. *Iconic displays* are a simple way to visualize data, which also prevents clutter.

[62] Dot Grid map example from Kartograph http://kartograph.org/showcase/dotgrid/

[63] Voronoi map example http://mbostock.github.io/d3/talk/20111116/airports-all.html

Dense Pixel displays and *geometric visualizations* like charts can be used to encode more complex information.

- **Icon-based, Glyphs**

  Matthew O Ward [**?**] defines *glyphs* as "graphical entities that convey one or more data values via attributes such as shape, size, color, and position". While the work of Otto Neurath on *ISOTYPE* [**?**] (1930s) can be seen as fundamental for *pictorial statistics*, the best-known literature reference to glyphs is "Chernoff faces" [**?**]. As (e) figure **??** indicates, data is encoded into properties of the face icon, such as shape of nose, mouth, eyes. Other fundamental glyph-based techniques include *stick figures* [**?**], *color icons* [**?**], *Hyperbox* [**?**] and *shape coding* [**?**]. Figure **??** extends this list by showing examples of glyphs that Ward collected for his taxonomy of glyphs placement strategies [**?**].

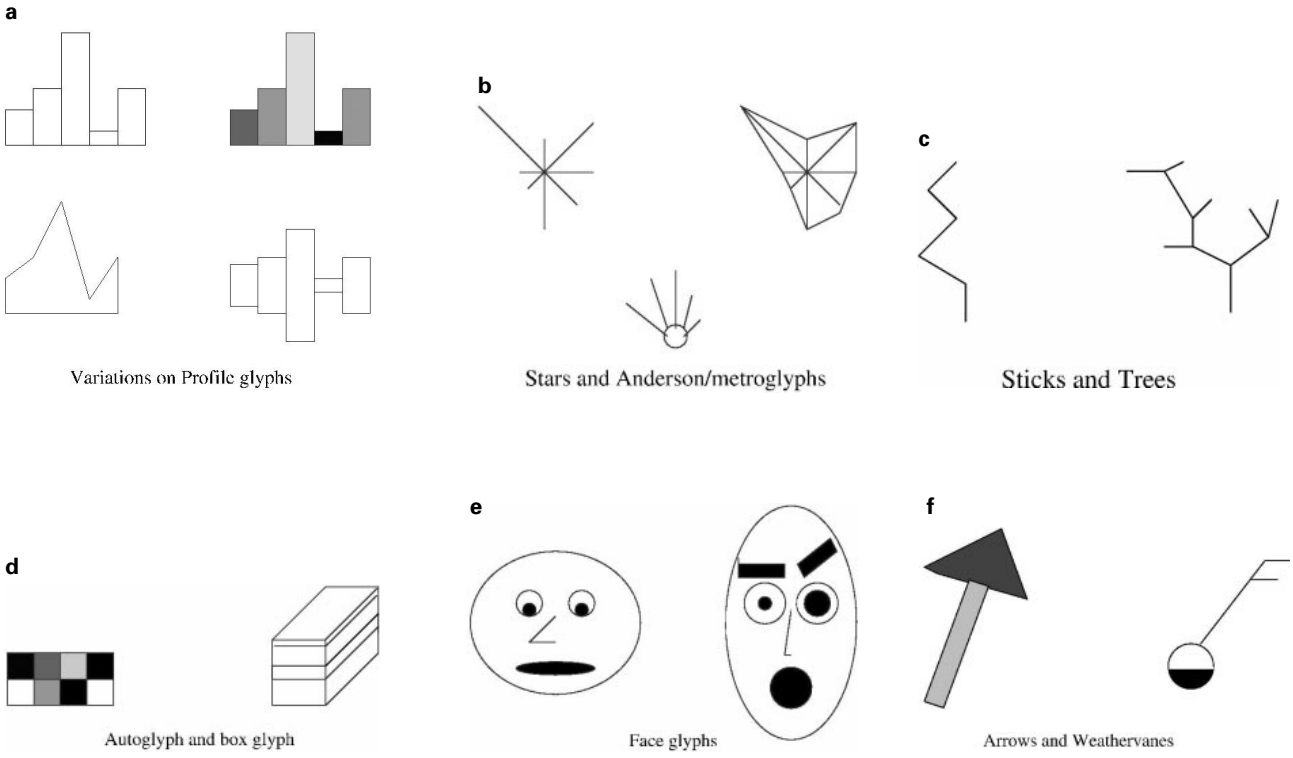


Figure 3.11: Examples of glyphs. Top row: (a) variations on profiles; (b) stars/metro-glyphs; and (c) stick figures and trees. Bottom row: (d) autoglyphs and boxes; (e) faces; and (f) arrows and weathervanes. [**?**].

Some glyph types have been created to identify clusters or similarities by plotting them side-by-side on a 2-dimensional plane, a technique which is referred to as *mosaic-based rendering*. *Stick figures* and *mosaic metaphors* are examples in that field [**?**, **?**]. One the other hand, reducing visual clutter as explained in chapter **??** also matters for glyphs, especially when putting them on a map. A trade-off between information-richness vs. simplicity and clarity has to be made. As Zhang writes, "with the amount of data increasing, the user hardly makes any sense of most properties of data intuitively, this is because the user cannot focus on the details of each icon when the data scale is very large" [**?**]. We can compare this to the map use cube presented in chapter **??**: more complex glyph types seem to be better suited for scientific purposes which can be related to private uses, while simpler glyph types seem more appropriate for presenting data to a public audience.

Examples for uses of simple icons and glyph types for clustered data on maps can be found in JavaScript mapping libraries as seen in chapter **??**. These are usually based on a simple icon or geometric shape like a circle or marker and use color coding and size variations as indicators for underlying information. Further examples are scaled data values and scaled dots [**?**], as well as the proportional symbol map [**?**]. In contrast to those presented so far, figure **??** visualizes eight simple glyphs [**?**].

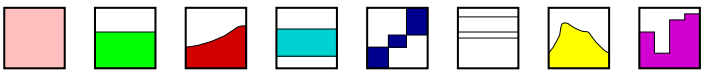


Figure 3.12: Eight different glyphs for aggregated edges (color shade, average, min/-max histogram, min/max range, min/max tribox, Tukey box, smooth histogram, step histogram) [**?**].

- **Pixel-oriented**

  Pixel-oriented techniques display the most possible information at a time by mapping each attribute value of data to a single, colored pixel. Color mapping approaches such as *linear variation of brightness*, *maximum variation of hue* and *constant maximum saturation* are used to color pixels which are arranged within limited space. By providing an overview of large amounts of data, pixel-oriented display techniques are suitable for a variety of data mining tasks in combination with large databases [**?**].

  The first pixel-oriented technique was presented by Keim [**?**] as part of the *VisDB system*. Large amounts of multidimensional data are represented as *Spirals* and

Axes. Figure **??** illustrates how a spiral would be constructed and figure **??** shows a rendered result of the axes technique. Further developments include the *Recursive Pattern Technique* [**?**] and the *Circle Segments Technique* [**?**]. Figure **??** visualizes such a circle which represents about 265,000 50-dimensional data items.

While no real-world examples have been identified during the research, using pixel-oriented techniques for visualizing complex clusters on a map seems possible. As the visualization relies on a large amounts of multidimensional data being present within clusters, the clustering algorithm would need to provide such required data. Performance implications also have to be considered, as potentially multiple clusters will be visualized on a map, sometimes even in real-time.

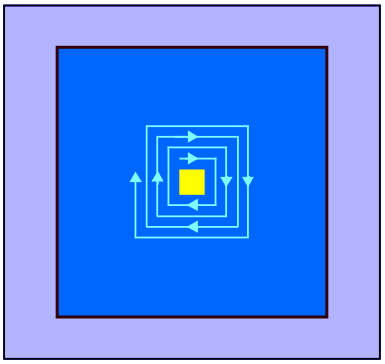


Figure 3.13: Spiral [**?**]

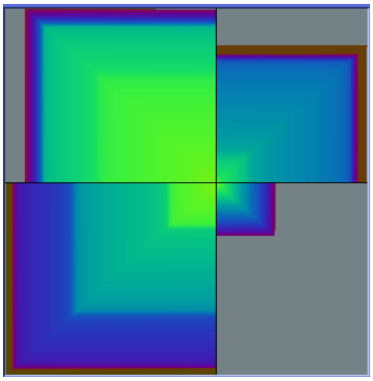


Figure 3.14: Axes [**?**]

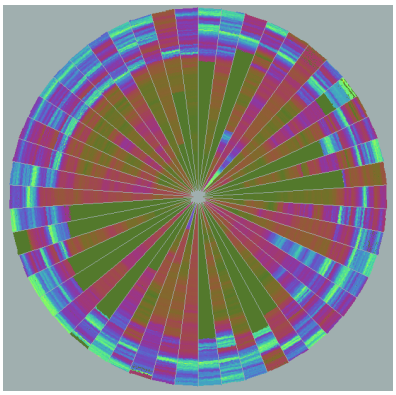


Figure 3.15: Circle [**?**]

- **Geometric techniques & Diagrams**

  Geometric techniques produce useful and insightful visualization by using geometric transformations and projections of the data. *Diagrams* are algorithmically drawn graphics that visualize data. This section lists a selection of geometric techniques for multivariate data presented by Ke-Bing Zhang [**?**], diagram types described by Dieter Ladenhauf [**?**] and related examples found in additional literature and on the web as stated in the individual references.

  - *Line charts* visualize data as lines by connecting data points of the corresponding values. They are used to display trends over time. Figure **??** illustrates surface temperature anomalies from NASA's GISS[64] as a *Sparkline* map [**?**]. The Sparkline is a reduced line chart without axes and coordinates. It presents the general shape of variation in a simple and highly condensed way [**?**].
  - *Bar charts* express data values by vertical or horizontal bars, in which the length of a bar indicates the data value. Figure **??** shows an example from

[64]NASA Goddard Institude for Space Studies `http://www.giss.nasa.gov/`

UgandaWatch[65], which displays economic indicators per region for the country Uganda. In this example, the Drupal mapping stack explained in chapter **??** is combined with bar chart technologies[66].

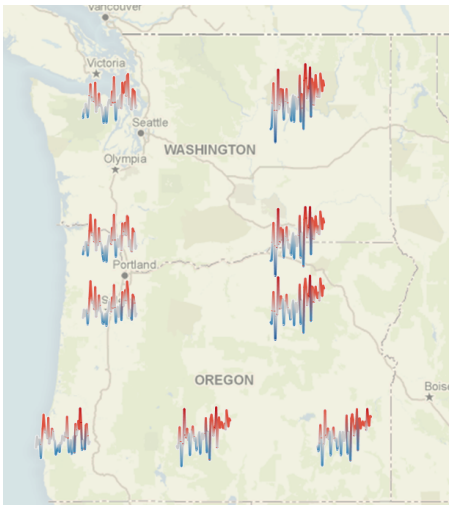


Figure 3.16: Sparkline map[67]

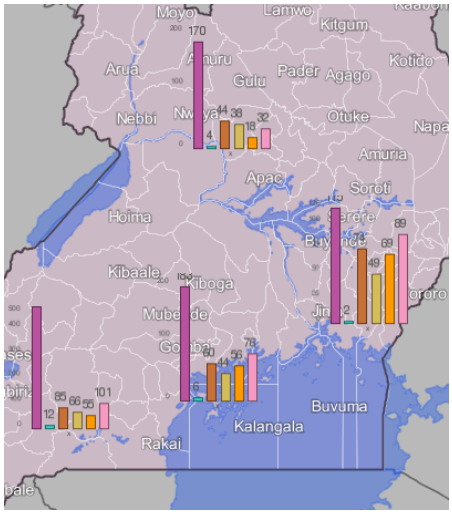


Figure 3.17: Bar chart map[68]

– *Pie charts* use a circle divided into sectors for expressing the proportional significance of data values. Variants of pie charts include *doughnut charts*, *three-dimensional pie charts* and *multi-level pie charts*. Also, the *polar area diagram* introduced in figure **??** is a special kind of pie chart and a further development of the *Bat's wing diagram* by Florence Nightingale [**?**].

  Figure **??** depicts a pie chart map example from the Kartograph[69] framework. It shows unemployment rates in Spain, providing an effective way to display ratios as opposed to a chloropeth map, where the user usually needs to consult a legend to understand the actual data values.

– *Container shapes* such as *bounding boxes* and *hulls* are an alternative to iconic displays as they can show the area covered by clusters [**?**]. As seen in figure **??**, the Leaflet.markercluster library visualizes the convex hull of a cluster to indicate the covered area on mouse-hover. Marco Cristani et al [**?**] use a hull-based technique for visualizing clusters from a geo-located image database on a map. As illustrated in figure **??**, each cluster is represented

[65] UgandaWatch: http://www.ugandawatch.org/

[66] How are you using mapping in Drupal? http://groups.drupal.org/node/174904#comment-585264

[67] Sparkline map example: http://www.tableausoftware.com/about/blog/2008/08/sparklines-maps

[68] UgandaWatch bar chart example: http://www.ugandawatch.org/

[69] Kartograph http://kartograph.org/

by a hull that marks the boundaries of the area and contains a representative image of the clustered set of images.

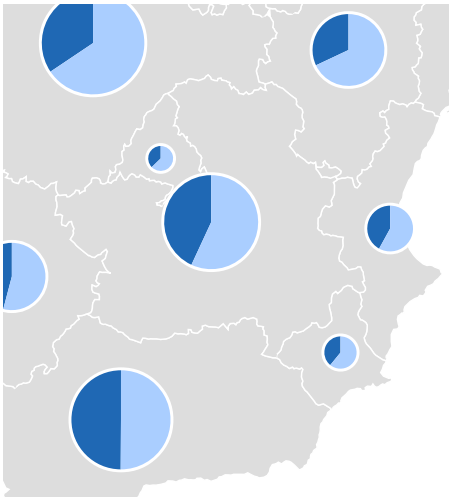


Figure 3.18: Pie chart map[70]

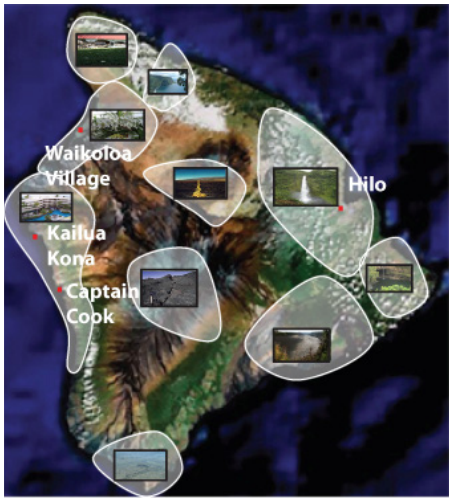


Figure 3.19: Convex hull map[71]

– Further chart types that can possibly used for visualizing clusters on a map include *Area charts* and *Star plots* [**?**] or more complex ones like *Parallel coordinates*, *Scatterplots*, *Treemaps* [**?**] or the *Contingency Wheel++* [**?**]. *Bristle maps* are an interesting approach for visualizing spatio-temporal data on a map. Basically, histograms of the data are rendered onto linear map elements. If the data permits a mapping from aggregates to spatial line data, such a visualization technique could be of interest [**?**].

This concludes the investigative enumeration of cluster visualization techniques for maps. It is by no means a complete, but rather an exemplary listing that may provide a starting point when researching visual means of presenting clustered data on a map.

An interesting publication by Andrienko et al [**?**] presents a complex system for place-oriented analysis of movement data. This goes beyond the use case of simply visualizing clustered data on a slippy map, but it is a good show case for how effective the presentation of spatio-temporal data using a combination of techniques can be. *Time graphs*, *mosaic diagrams*, *space-time cubes* and *table lens displays* are used to create a powerful tool for inspecting movement data on maps.

[70] Pie chart map example from Kartograph: http://kartograph.org/showcase/charts/
[71] Convex hull map example from [**?**]

3.4.3 Evaluation of visualization techniques for clusters on a map

This chapter summarizes the main visualization examples presented in the previous chapters. An evaluation of techniques for visualizing clusters on maps is created by proposing a set of key characteristics that are considered to be significant for this kind of visualization.

Evaluating information visualization techniques is a well-known problem. Undertaking an evaluation that is capable of "proving" the effectiveness is impossible in many situations as it would require too many tasks, data sets, implementations and users. Ellis and Dix state exploratory analysis as the most effective approach for evaluating visualization techniques [**?**, **?**]. In this sense, the following evaluation should primarily be treated as exploratory and as a help for understanding how cluster visualization on maps works, rather than a final, summative conclusion of which technique is superior than another.

Criteria. the following, custom criteria have been defined: *category*, *shows number of items within cluster* (*by shape size*, *by color or other*), *shows cluster area*, *shows extra cluster info (extra cluster info complexity*). Some criteria contain sub-criteria which are stated within parentheses.

In chapter **??**, three taxonomies have been introduced: *visual variables*, a *classification of visual data exploration techniques* and a *clutter reduction taxonomy*. An attempt to classify the different visualization techniques for representing clusters on maps according to classes introduced by these taxonomies didn't feel valid. Some criteria are too general while others are too specific to actually matter for visualizing clusters on maps, so the resulting data would not have much value. Similarly, a classification based on all *visual variables* would have become very complex. As the presented visualization examples describe general concepts, there are many possible variations that would increase the data set even more. It is still helpful to rely on the aesthetic attributes for a better understanding of how each visualization is constructed. Especially the *shape*, *size* and *color* attributes are considered to have a strong effect on visualizing clusters on maps and are therefore included within the evaluation.

An explanation of each criterion follows:

- **category**: Determines the type of visualization technique being evaluated. Possible values are *type of map* (see chapter **??**), a visualization *example*, as well as abbreviations for cluster visualization techniques (see chapter **??**): *glyph*: Icon-based, Glyphs, *pixel*: Pixel-oriented techniques and *geom*: Geometric techniques & Diagrams.
- **shape**: Defines the type of shape being used for the visualization of clusters on the map. For example *circle* or *area*. Refer to the visual variable shape in chapter **??**.

- **shows # of items within cluster**: If the visualization provides an indicator of the amount of items per clusters. This relates to the 'can see overlap density' criterion of the clutter reduction taxonomy, see **??**. Two sub-criteria are used to differentiate between visual means of encoding the number of items within clusters:
  - **by shape size**: classifies visualization techniques that use the shape size for indicating the number of items within clusters.
  - **by color or other**: classifies visualization techniques that use color or other visual indicators to describe the number of items within a cluster.
- **shows cluster area**: Determines, if the visualization indicates the spatial area that is covered by the cluster or the items within a cluster.
- **shows extra cluster info**: Besides the two characteristics of number of items within a cluster and the cluster area, the technique might provide means of visualization additional information of clusters such as aggregates.
  - **extra cluster info complexity**: This sub-criterion expands of an intuitive notion of complexity that can be visualized as extra cluster info by the technique. *low* indicates a maximum of three dimensions. *medium* is used to describe up to 12 dimensions of additional data and *high* classifies cluster visualization techniques that go beyond this number of dimensions.

The results of the evaluation of visualization techniques for clusters on a map based on the stated criteria are illustrated in figure **??**. Note that the classifications for each technique being evaluated primarily represent the according examples shown in the previous chapters. Where it seemed obvious, the optional possibility of fulfilling a criterion has been marked as such. As an extreme example, the geographic map as its general concept has been marked with the optional possibility of fulfilling each criterion. It is up the the actual implementation to satisfy them individually. While the Leaflet.markercluster example provides a visual indicator for the number of items within clusters, the wind history example doesn't.

Some classifications contain a number referring to additional notes as presented in the following: (1) The Leaflet.markercluster example shows cluster on demand, based on user interaction. By mouse-hovering over an item, it will display the convex hull of the items being clustered, see figure **??**. (2) In the case of the binned heat map example and the Voronoi map, cluster areas are approximated by the tessellation which is part of the clustering algorithm, see figure **??** and **??**. (3) The dot-grid map doesn't provide a mean of showing cluster areas by themselves, but the density of items still supports the notion of recognizing cluster areas, see figurefig:map-type-dotgrid. (4) For various

	category	shape	shows # of items within cluster	by shape size	by color or other	shows cluster area	shows extra cluster info	extra cluster info complexity
Geographic map	map type	~	~	~	~	~	~	~
Leaflet.markercluster	example	circle	x	-	x	~ (1)	-	-
Wind history	example	polar area diag.	-	-	-	-	x	low
Choropleth	map type	area	x	-	x	x	~	low (7)
Heatmap	map type	area (hexagon)	x	-	x	~ (2)	~	low (8)
Dot-grid map	map type	circle	x	x	~	~ (3)	~	low (8)
Voronoi	map type	area (convex)	~	~	~	~ (2)	~	low (7)
Face glyph	glyph	ellipsis	~ (4)	~ (4)	-	-	x	medium
Step histogram glyph	glyph	rectangle	~ (4)	~ (4)	-	-	x	low
Spiral	pixel	spiral	~ (4)	~ (4)	- (6)	-	x	high
Axes	pixel	rectangle	~ (4)	~ (4)	- (6)	-	x	high
Circle	pixel	circle	~ (4)	~ (4)	- (6)	-	x	high
Spark line	geom	line	~ (4)	~ (4)	- (6)	-	x	low
Bar chart	geom	rectangles	~ (4)	~ (4)	- (6)	-	x	low
Pie chart	geom	circle	x	x	- (6)	-	x	low
Hull	geom	area (hull)	~ (5)	-	~ (5)	x	x	low

Figure 3.20: Evaluation of visualization techniques for clusters on a map. Legend: 'x': yes, '~': possibly, '-': no. Numbers in parentheses reference additional notes within the accompanying text.

visualization techniques, the amount of items within a cluster could be visualized by simply scaling the visual entity. (5) The hull example uses an area shape defined by the data, similarly to the choropleth map. Without a distortion technique, the shape therefore can't be used to indicate the number of items within a cluster. Still, a non-shape visual aspect like color could be used as an indicator. (6) In the case of pixel-oriented techniques and the provided chart examples, the color attribute will likely be used for showing extra cluster info instead of indicating the number of items within a cluster. Modifying the shape size can be used as an alternative in this cases. (7) The choroleth and voronoi map examples would rely on representing extra information within the defined area and therefore rely on the.variation of visual variables related to color and texture. (8) The same restrictions as in the previous note apply, but for even smaller areas.

Driving forces in visual mapping, map visualization types and cluster visualization techniques for maps have been introduced and summarized within the given evaluation. This concludes the chapter on state of the art.

CHAPTER 4

Objectives

4.1 Performant real-time clustering

The main purpose of this thesis is to design and implement an algorithm that allows to create performant, scalable maps with Drupal by using server-side clustering in real-time. The algorithm needs to dynamically cluster geospatial data on the server-side, before it is rendered by Drupal and gets transferred to the client. As a result, the client-side mapping visualization component receives a limited amount of clustered data which can be processed and visualized efficiently enough to produce a smooth end-user experience.

The expected performance benefits of using a server-side geo clustering component to be designed and implemented for Drupal are:

1. Better server performance by only processing (pre-)clustered items
2. Better network performance by only transferring clustered items
3. Better client performance by only processing and visualizing clustered items

The goal is to build upon existing cluster theory, the current state of the art and the existing Drupal mapping capabilities. The following requirements apply for a successful clustering implementation:

- Cluster in real-time to support dynamic queries
- Cluster up to 1,000,000 items within less than one second.

4.2 Visualization & Usability

Clustering data on maps not only affects performance, it also changes the way, the user will see and interact with the clustered data. Ideally, the clustering process should support the user in the task of exploring a large data set on the map by compacting the amount of information that is visualized. The way, how the clustered data is visualized on the map needs to communicate essential information about the clustered data like the size of a cluster. In addition, the user needs means of interacting with the clustered data being presented. The user should be able to reveal the details of clustered data for example by zooming in.

4.3 Integration and extensibility

The server-side clustering implementation should be designed for integration and extensibility. Integration should be provided or at least be possible with key components of the existing ecosystem for creating interactive-maps with Drupal as explained in **??**. There is also a need for means of extensibility within the clustering solution to facilitate further improvements of the clustering implementation.

The intended benefits of an integrated and extensible approach for the server-side clustering solution are:

- Integrate the clustering with JavaScript mapping libraries as Leaflet or OpenLayers.
- Integrate the clustering with Drupal and Apache Solr search backends.
- Allow to extend the clustering for adding alternative algorithms.

4.4 Open source

One of the main reasons for the wide adoption of Drupal as a content management system and framework is its licensing under the terms of the GNU General Purpose License (GPL). Being free and open source software gives any user the freedom to run, copy, distribute, study, change and improve the software. The intended server-side clustering solution would build upon the Drupal system and a number of extension modules essential to the creation of interactive-mapping solutions. Not only as a logical consequence, but also as a primary factor of motivation, the results of this thesis and

in particular the clustering implementation should be released under the free and open source GPL license.

An open process of planning, designing and developing a server-side clustering solution is intended to bring a number of benefits in contrary to a proprietary, closed source approach:

- The ability to discuss ideas and incorporate feedback from the community during the planning phase.
- The possibility for other community members to review prototypes and look at the source code.
- The potential for test results submitted by other community members, testing the solution.

4.5 Use cases

The practical use case for server-side geo clustering should add spatial search capabilities to the *Recruiter* job board solution.

> Recruiter is a Drupal distribution for building Drupal based e-recruitment platforms. Users can register either as recruiter and post job classifieds or they can register as applicants and fill out their resume. A faceted search helps users to find jobs and possible job candidates.[1]

Adding server-side geo clustering capabilities would allow to visualize several thousands of available jobs on an interactive map for large-scale e-recruitment websites. The server-side clustering solution should be designed for the possibility to be added to geospatial searches realized in combination with the Recruiter distribution. This influences the integration and extensibility requirements, stated in the previous chapter.

[1] http://drupal.org/project/recruiter

CHAPTER 5

Realization

This chapter describes the realization of server-side geo clustering for Drupal. First, an analysis based on the objectives stated in the previous chapter considers their implications for the implementation. Subsequently, the Geohash-based clustering algorithm is defined. A Geohash-based hierarchical, spatial index will be designed, as well as the actual clustering algorithm. Finally, the architecture and implementation of the algorithm for Drupal are explained in detail.

5.1 Analysis

Two objectives stated in chapter **??** are 1) *performance* and 3) *integration and extensibility*. Together, they define the target of creating an integrated solution for server-side clustering in Drupal with a clear focus on enhancing the performance of data-intense maps.

5.1.1 Algorithm considerations

By the definition of clustering foundations explained in **??**, a number of factors need to be considered when designing the clustering algorithm. For the clustering task, a *pattern representation* method, a *proximity measure* and the *clustering algorithm* itself need to be defined. In addition, the *cluster type* and the right choice of *clustering techniques* have to be considered. The following enumeration discusses the considerations for the named factors.

- **Pattern representation**: A number of spatial data types as points, lines or rectangles exist (see chapter **??**). For practical and simplicity reasons, only points as the simplest spatial data type need to considered for the pattern representation of the server-side clustering task. The exact selection of the pattern representation depends on the Drupal integration consideration, outlined in the following chapter **??**.

- **Proximity measure**: Amongst the options for proximity measures (see chapter **??**), the Euclidean distance is the obvious choice for the intended clustering task. What needs to be considered in this case though, are the implications of a map projection being used to represent the spherical geoid on a planar map on the computer screen (see **??**).

- **Cluster type**: Different means for the definition of cluster types have been explained in chapter **??**. The attributes of *well-separated*, *prototype-based* and *density-based* clusters seem logical for clustering points on a map. Most importantly, clusters should be defined by the centroid of all clustered items as their prototype. The attributes of the two other cluster types only apply to a certain extend. Well-separated ensures that clusters don't overlap which enhances readability of the map. Density-based clusters account for a visual grouping of items in crowded regions. For simplicity and readability, prototype-based clusters represented as single map markers are preferred over potentially polymorph well-separated or density-based clusters.

- **Clustering algorithm**: In order to support dynamic queries, the clustering task needs to be performed on-the-fly. On the other hand, the clustering should perform efficiently. Chapter **??** explains how the grid-based STING algorithm precalculates clusters in order to achieve a constant time complexity for the actual retrieval of cluster items at query-time. The intended design and implementation of the server-side clustering algorithm needs find a good balance between performance while still guaranteeing an on-the-fly clustering of dynamically retrieved data sets.

5.1.2 Drupal integration considerations

Drupal already provides a variety of tools in order to create interactive maps. The following chapter analyses how a server-side clustering implementation could integrate with existing Drupal mapping tools that have been explained in chapter **??**.

- **Configuration**: Most Drupal modules provide a user interface that helps configure settings. Similarly, the server-side clustering implementation should be con-

figurable using a user interface in order to setup and parametrize the clustering process. Drupal 7 includes a versatile *Form API*[1] that helps build such forms and potentially should be used to achieve this requirement.

- **Storage**: The de-facto standard for storing geospatial data in Drupal 7 is the Geofield module. By default, it doesn't provide a *spatial index* but stores spatial data in the database field table as separate columns for latitude, longitude and other related spatial information as the bounding box. Given the popularity of the Geofield module, it should be considered as the primary source for spatial data to be processed within the server-side clustering implementation.

  The Recruiter distribution uses the Search API module suite for performant queries using the Apache Solr search platform. In order to integrate the server-side clustering solution with Recruiter, a possibility for indexing the spatial data using in Solr using the Search API module has to be found.

- **Querying**: Drupal integration on the query level primarily needs to happen in combination with the Views and Views GeoJSON modules. The clustering task therefore needs to be integrated into the process of how the Views module queries data and processes its results. The Views module provides an extensive API[2] that allows to extend its functionality.

  Two main challenges of integrating a server-side clustering solution with Views have been identified: 1) *allow to inject a custom aggregation implementation*[3] and 2) *dealing with geospatially clustered data*[4]. The first challenge deals with finding a clean way to integrate a clustering solution into the processing queue of the Views module. The second subsequently deals with challenges that arise when processing and visualizing the clustered data afterwards. As the clustering process changes the data being processed, the implementation needs to take care of involved APIs that work with the changed data.

  Similarly to the previously discussed storage aspect of the Drupal integration, the querying component to account for clustering data in combination with Solr and the Search API.

- **Visualization**: Once clustered, the data needs to be visualized on the client. Requests based on the Bounding Box strategy in combination with a JavaScript mapping library will supply the clustered data for the client. The clustered data then needs to be visualized appropriately.

[1] http://api.drupal.org/api/drupal/developer!topics!forms_api_reference.html/7

[2] http://api.drupal.org/api/views

[3] http://drupal.org/node/1791796

[4] http://drupal.org/node/1824954

Based on the previous insights regarding the algorithm and Drupal integration considerations, a concrete clustering algorithm needed to be found. As explained, multiple data storage backends including the default MySQL-based Geofield storage and Apache Solr as indexing server are considered which means that the clustering algorithm should not rely on a particular database. The Geohash encoding algorithm for spatial coordinates into string identifiers has been explained in chapter **??**. Its hierarchical spatial structure can be leveraged as a spatial indexing mechanism that is abstracted from the concrete database implementation. In the following, the concrete algorithm, architecture and implementation will be explained.

5.2 Geohash-based clustering algorithm

The clustering algorithm leverages a hierarchical spatial index based on Geohash. In a *first step*, it will initialize variables for the clustering process. The *second step* creates an initial clustering of the data set based on Geohash. In a *third step*, the agglomerative clustering approach merges overlapping clusters by using an iterative neighbor check method. Besides the actual data set, the clustering algorithm uses two main *input parameters*: the current zoom level of the map being viewed and a setting for the minimum distance between clusters. In the following chapter, the spatial index, the cluster definition and the clustering algorithm itself will be explained.

A **Geohash-based hierarchical spatial index** is created to support an efficient clustering process. For each location point, the latitude and longitude values are encoded as Geohash strings. Based on the gradual precision degradation property of Geohash, prefixes of the resulting string identifier for each length from 1 to the maximum Geohash length are stored separately. Each Geohash prefix is the identifier of the encoded point on the corresponding level in the spatial index hierarchy.

City	Latitude / Longitude	Geohash	Level 1	Level 2	Level 3	Level 4
Vienna	48.2081743, 16.3738189	u2ed	u	u2	u2e	u2ed
Linz	48.2081743, 16.3738189	u2d4	u	u2	u2d	u2d4
Berlin	52.5191710, 13.4060912	u33d	u	u3	u33	u33d

Table 5.1: Example of a Geohash-based hierarchical, spatial index.

Table **??** demonstrates an exemplary Geohash-based spatial index consisting of three European cities: Vienna, Linz and Berlin. All three cities share the Geohash prefix of length one, while only Vienna and Linz lie within the same Geohash prefix of length two. None of the cities share the Geohash prefixes of length three and four.

```
Input: unclustered geo data, points
       current zoom level, zoom
       minimum cluster distance, distance
begin Phase 1: initialize variables
    level ← getClusterLevel (zoom, distance);
    clusters ← ∅;
end
begin Phase 2: pre-cluster points based on Geohash
    for p ∈ points do
        prefix ← getGeohashPrefix (p, level);
        if prefix ∉ clusters then
            clusters ← clusters ∪ initCluster (p);
        else
            addToCluster (clusters.prefix, p);
        end
    end
end
begin Phase 3: merge clusters by neighbor-check
    for c1 ∈ clusters do
        neighbors ← getGeohashNeighbors (c1, clusters);
        for c2 ∈ neighbors do
            if shouldCluster (c1, c2, distance) then
                mergeClusters (clusters, c1, c2);
            end
        end
    end
end
Output: clustered results, clusters
```

Algorithm 5.1: K-means algorithm [**?**]

For the particular clustering algorithm, a **cluster** is defined as a spatial point based on one or many clustered points. The location of the cluster is defined by the latitude and longitude values of the centroid of its sub-points. Besides its location, a cluster also needs to store the number of items contained within the cluster. In addition, a cluster may contain optional, aggregated information about the clustered items as a list of their unique identifiers or references to the original items.

The **Geohash-based algorithm** takes advantage of the spatial index create cluster items efficiently. A prototypical pseudo-code is provided at algorithm **??**. In the following section, the input, its 3 phases and the output of the clustering algorithm will be dis-

cussed.

- Three *input parameters* are required by the clustering algorithm: *points* is the set of geo data item to be clustered. These points need to be indexed based on the previously described *Geohash-based hierarchical spatial index*. In addition, *zoom* describes the zoom level of the map being viewed and *distance* is the minimum distance between clusters in pixels. Together, the zoom and distance parameters allow to control the granularity of the clustering process.

- **Phase 1: initialize variables**

  In the initialization phase, the clustering algorithm prepares for the actual clustering process. Primarily the clustering *level* is computed by a *getClusterLevel* function. This function needs calculate a clustering *level* so that clusters of reasonable sizes are generated the given *zoom* and *distance* parameters.

- **Phase 2: pre-cluster points based on Geohash**

  Clusters are created by aggregating all points that share a common Geohash-prefix of length equal to the clustering *level* into preliminary clusters.

  For each point, the *getGeohashPrefix* function determines a *prefix* Geohash-prefix is determined, essentially accessing the prefix of length *level* from the *Geohash-based hierarchical spatial index*. In an agglomerative process, clusters are created for all *points* that share the same *prefix*. If for a given *prefix* no *cluster* exists at the moment, the *initCluster* function initializes a new cluster based on the point. If the *prefix* already has been added as cluster, the *addToCluster* function adds the point to the appropriate, existing cluster and updates cluster information as the centroid and the number of items within the cluster.

  The preliminary clustering based on Geohash is suboptimal, because of the edge cases described in chapter **??** and illustrated in figure **??**. In order to account for overlapping clusters at the edges of Geohash cells, a third phase merges clusters by a neighbor-check.

- **Phase 3: merge clusters by neighbor-check**

  Overlapping neighbor clusters are merged in the third phase of the clustering algorithm. For every cluster in the precomputed *clusters* from phase 2, a *getGeohashNeighbors* function determines all neighbors relevant for the neighbor check. A *shouldCluster* function determines, if two clusters need to be merged based on the *distance* parameter compared to their relative *distance*. It needs to implement a *Euclidian distance* measure as explained in chapter **??** but also account for implications of the roughly spherical geoid and map projection introduced in chapter **??**. If two

clusters should be merged, a *mergeClusters* function joins them together similarly to the *addCluster* function described in phase 2.

- The *output* of the Geohash-based clustering algorithm is a set of *clusters* that satisfies the minimum cluster distance criterion specified by the *distance* parameter.

The described Geohash-based algorithm is grid-based and can be compared to the STING algorithm described in chapter **??**. The STING algorithm precomputes clusters to achieve constant time complexity relative to the number of grids. By the requirement to compute clusters in real-time, this approach was out of scope for the Geohash-based algorithm. Instead, it leverages the grid to achieve a time complexity is linear to the number of points O(n).

5.3 Architecture & Implementation

Geocluster is a Drupal 7 module that implements the Geohash-based clustering algorithm. It has been developed as the practical part of this thesis and published under the GPL-license on drupal.org:

`http://drupal.org/project/geocluster`

An iterative approach was taken in order to explore ways to fulfill the various integration and extensibility requirements formulated in chapter **??** and analyzed in chapter **??**. Based on the Drupal mapping stack explained in chapter **??**, a high-level architecture for implementing the Geohash-based clustering algorithm was designed.

5.3.1 Principles

Geocluster has been implemented following a number of principles:

- Leverage existing APIs and *hooks* when possible
- Program object-oriented code where it makes sense
- Implement changes to existing modules as *patches* if necessary

5.3.2 Architecture overview

The parts involved in the Geocluster system are

- Integration of the Geohash-based hierarchical spatial index with Geofield
- Server-side clustering implementation
  - Configuration of the clustering process
  - Integration of the clustering process with Views
  - Implementation of the clustering algorithm
- Client-side Geocluster Visualization component

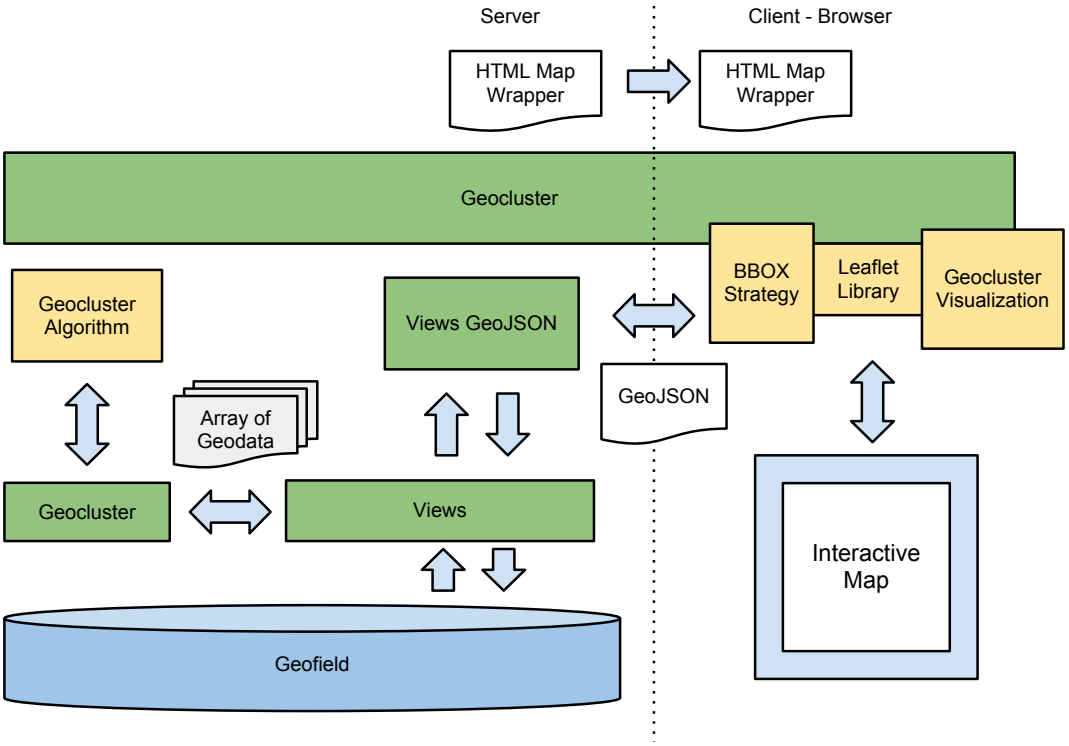


Figure 5.1: Geocluster architecture overview.

Figure **??** depicts how the Geocluster module integrates with other components of the Drupal mapping stack. The main point of integration for the Geocluster module on the

server-side is the Views module. When the Views module performs a spatial query that has been configured for clustering, Geocluster will integrate the clustering algorithm into the Views execution process. The clustering task may interact with the Views module *before* and *after* a query has been executed. Different algorithm implementations may rely on modification of the Views query while others only need to post-process the results of a Views query. After the clustering process has been finished, the server-side execution process continues as usual. The clustered result data is rendered, for example as GeoJSON feed using the Views GeoJSON as indicated in the diagram. On the client-side, a Geocluster Visualization component is used to properly visualize clustered results. The Geocluster module therefore integrates with the Leaflet and Leaflet GeoJSON modules in order extend the Bounding-Box driven communication of clustered results between client and server.

5.3.3 Integration of the Geohash-based hierarchical spatial index with Geofield

The Drupal Field API that Geofield uses has been leveraged to spatially index locations stored by the Geofield module. The field schema for fields of the Geofield type is extended by *geocluster_field_schema_alter* to add separate columns for the Geohash indices that form the spatial index. The *geocluster_field_attach_presave* hook implementation takes care of saving the additional index information whenever a Geofield location value is saved to the database.

During the development of the Geocluster module, support for encoding location values into Geohash has been added to the geoPHP library[5]. Subsequently, a patch to make use of geoPHP's Geohash support has been created for the Geofield module and committed[6].

5.3.4 Server-clustering implementation

The server-side clustering implementation consists of three parts: configuration of the clustering process, integration of the clustering process with Views and implementation of the clustering algorithm.

Figure **??** illustrates how the *GeoclusterAlgorithm* allows for different variations of the clustering algorithm to be implemented and how the configuration integrates with the Views module.

[5] https://github.com/phayes/geoPHP/issues/32
[6] http://drupal.org/node/1662584

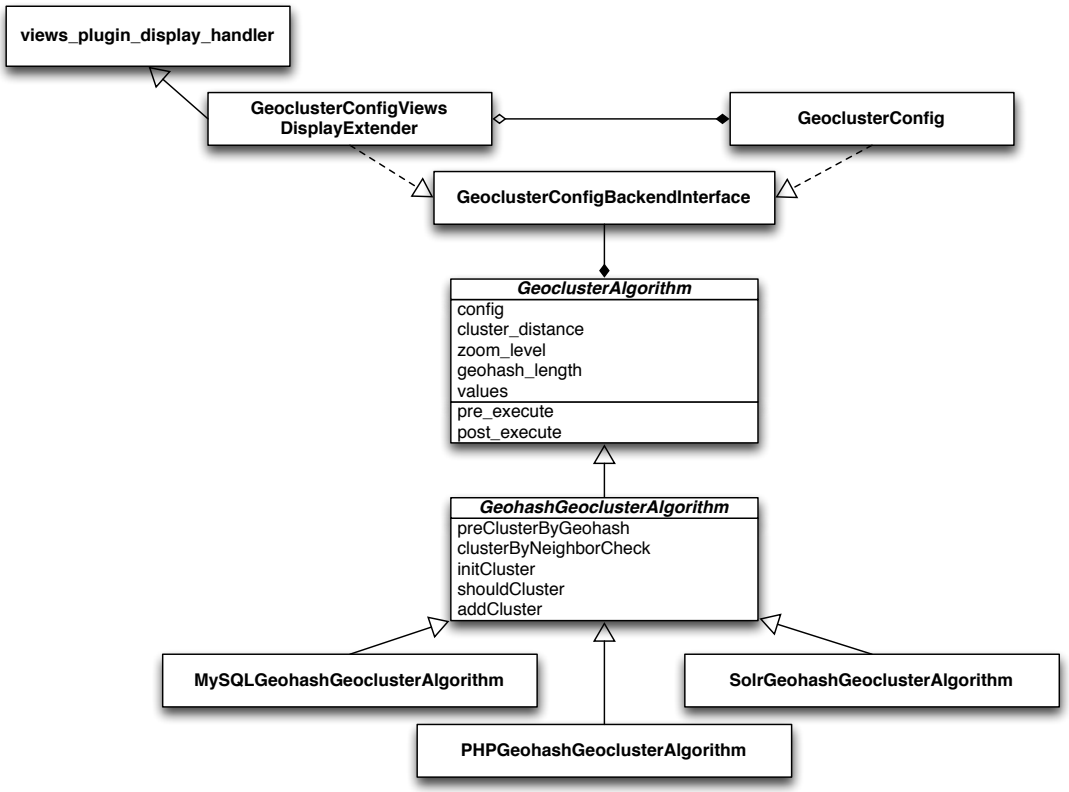


Figure 5.2: Geocluster class diagram.

5.3.5 Configuration of the clustering process

The Geocluster algorithm depends on three inputs: the data points to cluster, the current zoom level and the minimum cluster distance. The zoom level is passed by the bounding box strategy as a request parameter. In order to configure the rest of the inputs required by the algorithm, a user interface for configuration of the clustering process has been created using the Views plugin system.

When configuring a View, the user may enable Geocluster using a checkbox as illustrated in figure **??**. If enabled, an additional option set for the clustering process will be displayed.

- The **Clustering algorithm** option allows the user to select one of the provided clustering algorithms.

- The **Cluster field** option determines the Geofield which should be used as spatial data source for the clustering process.
- The **Cluster distance** option specifies the minimum distance between clusters for the algorithm.

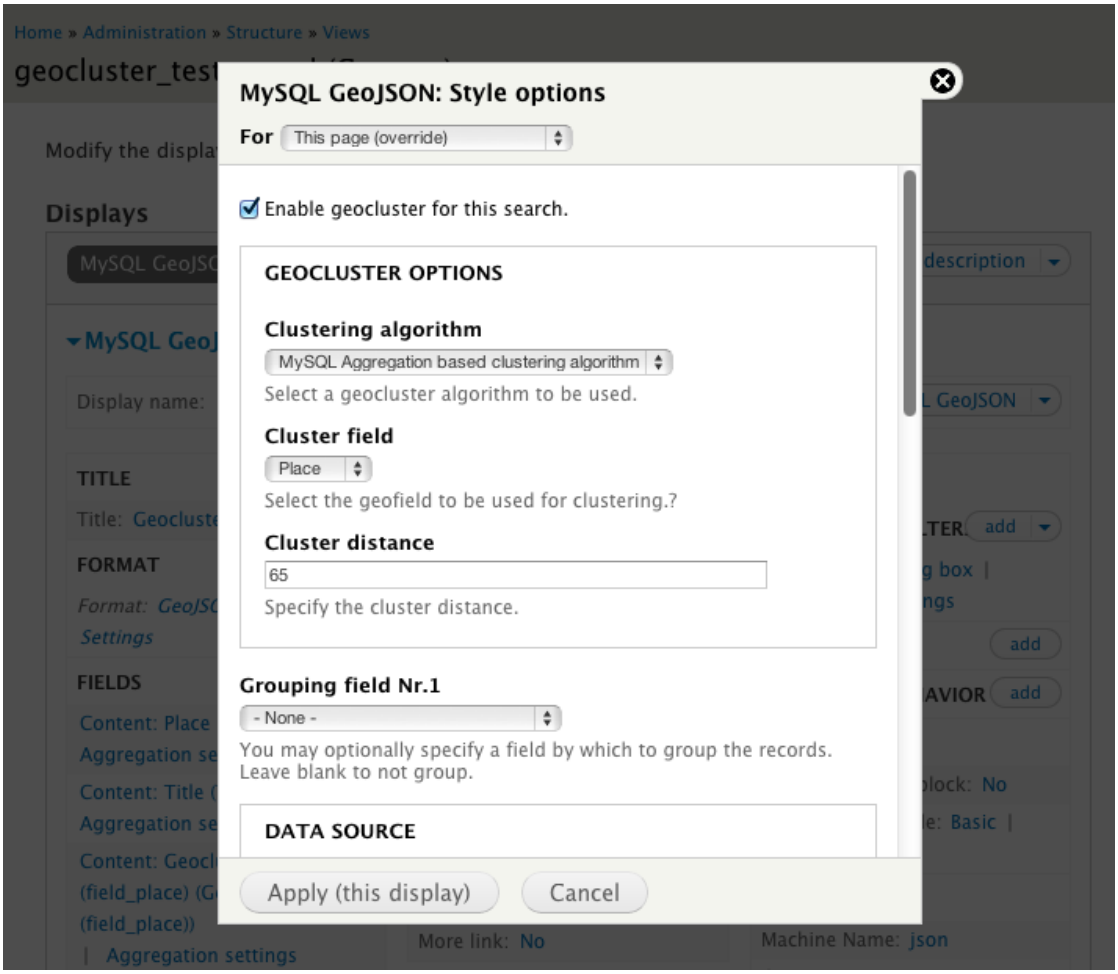


Figure 5.3: Geocluster configuration options within the Views administration interface.

As indicated by figure **??**, the *GeoclusterConfigViewsDisplayExtender* class integrates the configuration options as a Views plugin. The actual configuration options have been decoupled from the Views-dependent plugin as the *GeoclusterConfig* class. The interface *GeoclusterConfigBackendInterface* abstracts the communication between the configuration classes and the *GeoclusterAlgorithm*.

5.3.6 Implementation of the clustering algorithm

Three variations of the Geohash-based algorithm have been implemented for Drupal 7. In a first iteration, a *PHP-based* implementation of the clustering algorithm was prototyped to figure out the integration of the algorithm into the Drupal mapping stack. In a second and third iteration, *MySQL-aggregation-based* and *Solr-based* algorithm implementations were added to improve performance and support additional use cases.

The abstract *GeoclusterAlgorithm* class defines the basics of a clustering algorithm. A constructor initializes the clustering task according to phase 1 of the algorithm. The algorithm base class provides access to the configuration options for the algorithm. In addition, it defines *pre_execute* and *post_execute* as the two main methods that will allow the actual algorithm implementation perform its clustering task.

A second, abstract *GeohashGeoclusterAlgorithm* class encapsulates common infrastructure for the Geohash-based clustering algorithms. An empty stub for creating an initial clustering by using the geohash grid is defined as *preClusterByGeohash*, as well as a default implementation of *clusterByNeighborCheck* that creates final clusters by checking for overlapping neighbors. Further methods are common helper function defined by the algorithm as *initCluster*, *shouldCluster* and *addCluster*.

The actual implementations of the geocluster algorithm are realized as plugins using the CTools plugin system[7]. This allows other modules to implement their own algorithm plugins which will automatically be exposed within the geocluster configuration options. As an example, the Apache Solr-based geocluster implementation has been implemented within a separate, optional sub-module *Geocluster Solr*.

- **PHPGeohashGeoclusterAlgorithm** is an exemplary, PHP-based implementation of the Geohash-based clustering algorithm. It was primarily created as a first prototype to demonstrate clustering functionality, test the algorithm and work out integration issues with Drupal.

  The PHP-based algorithm is completely decoupled from the database, as it only relies on Geofield and performs all clustering logic from phases 2 & 3 in a the post-execution step of the algorithm. On the other hand, this also makes it the least performant algorithm implementation, because the entire set of results has to be loaded before the clustering process is executed.

  PHPGeohashGeoclusterAlgorithm uses an adapted version of the *views_handler_field_field::post_execute()*[8] method. Its intention is to use

[7] http://drupal.org/project/ctools

[8] http://api.drupal.org/api/views/modules!field!views_handler_field_field.inc/function/views_handler_field_field%3A%3Apost_execute/7

field_attach_load[9] to load just the necessary field information instead of loading items whole entities before the clustering process.

- **MySQLGeohashGeoclusterAlgorithm** is a MySQL-aggregation-based implementation of the clustering algorithm. Its intention is to improve performance by shifting the time-critical part from phase 2 of the clustering process into the database. In comparison to the post-execution based PHP algorithm, the database query delivers an already pre-clustered result. The neighbor-check of the algorithm is then performed on the pre-clustered result from the database.

  The Geohash-based pre-clustering is realized as a combination of the *GROUP BY* clause with *aggregate* functions. The pre-execution step of the algorithm adds the clustering-specific aggregation settings to the query. The results will be grouped by the column that matches the index level determined by the clustering initialization step. In addition, a the $COUNT$ function is used to calculate cluster sizes and AVG provides an approximation of each cluster's centroid.

- **SolrGeohashGeoclusterAlgorithm** is a Search API Solr-based implementation of the clustering algorithm. It improves performance by shifting the time-critical part from phase 2 of the clustering process into the Solr search engine, similarly to the approach of the MySQL-based algorithm.

  Clustering with Solr requires the Geohash-based spatial index to be integrated with the Solr search engine. Also the execution process of a Search API based view differs from the default Views execution process in a number of ways. Finally, the results being processed in a Sarch API View have a different structure from the default result structure of a views result. The resulting complexity of integration of Geocluster with Search API Solr motivated the creation of a separate sub-module *Geocluster Solr* that contains necessary infrastructure and helpers.

  Figure **??** visualizes how Geocluster Solr is integrated into the default architecture of Geocluster, as described in figure **??**. The original intention was to create a Solr plugin that would perform the entire algorithm within the Solr search engine. A draft for such a plugin has been created on github[10]. In order to facilitate the use of Geocluster Solr without the need for installing an custom Solr plugin and for the lack of understanding of the Solr API, a simpler approach was taken. Instead of performing the entire clustering process within Solr, just the first step of creating clusters based on Geohash is realized using a standard Solr query. This still keeps the time critical task within Solr.

  A custom search service class *GeoclusterSearchApiSolrService* has been defined that implements the main clustering logic within a *preQuery* and a *postQuery*

[9] http://btmash.com/article/2012-04-13/i-just-want-one-field-using-fieldattachload
[10] https://github.com/dasjo/SolrGeocluster

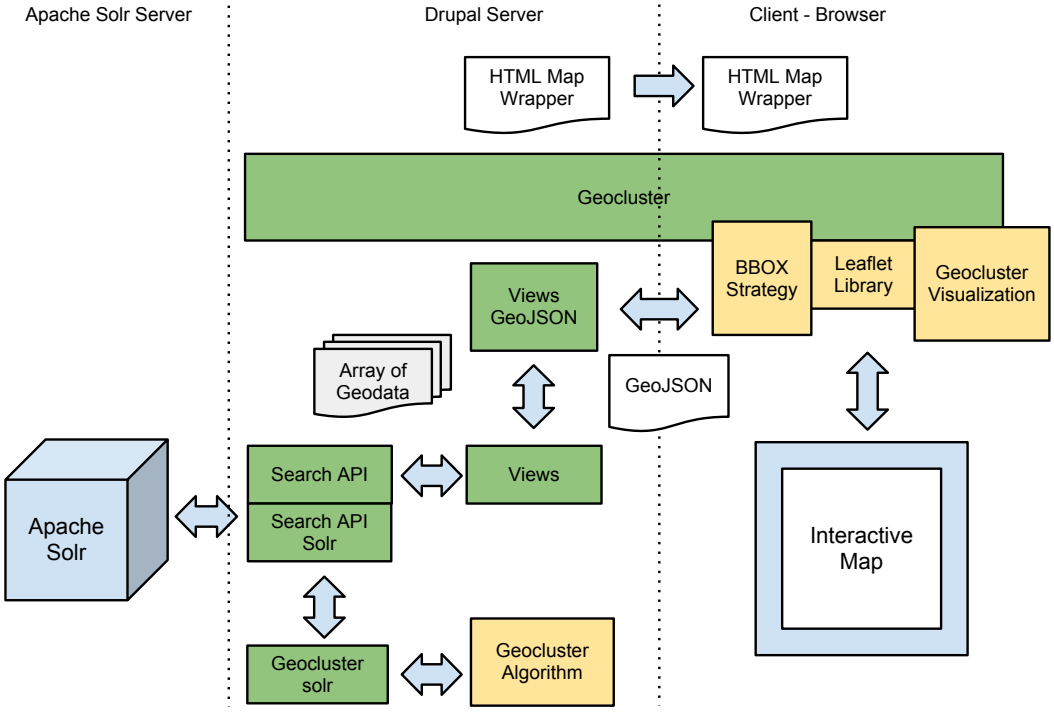


Figure 5.4: Geocluster Solr architecture overview.

method. Within the preQuery step, the Geohash-based pre-clustering step is configured by using the *result grouping*[11] feature of Apache Solr. The query is configured to return groups of results based on the clustering index level. The postQuery step maps the Solr-based results into a processable structure and delegates to the generic *clusterByNeighborCheck* method of the clustering algorithm.

Two helper classes support the clustering task. *GeoclusterHelper* provides a set of geospatial PHP functions to in order to calculate the distance between two points on the map in pixels based on the zoom resolution and the haversine formula[12]. Additional helpers support coordinate system conversions for the Spherical Mercator projection, see chapters **??** and **??**. *GeohashHelper* helps initializing the algorithm by a *lengthFromDistance* function that determines the appropriate geohash prefix length based on zoom level and minimum distance in pixels.

[11] http://wiki.apache.org/solr/FieldCollapsing
[12] http://en.wikipedia.org/wiki/Haversine_formula

5.3.7 Client-side Geocluster Visualization component

A simple Geocluster visualization component has been built to support the display of clustered markers on interactive maps based on the output of the server-side clustering implementation. It extends the Bounding Box strategy of the Leaflet GeoJSON module in oder to create numbered markers that visualize the cluster sizes. Clicking on a clustered marker will zoom into the map in order to explore the data on a more granular level. Figure **??** demonstrates an example screenshot of the Geocluster visualization. Compare this with an unclustered map, containing the same amount of items in figure **??**

Figure 5.5: Geocluster visualization: a Leaflet map containing clustered markers.

The Bounding Box related logic for Leaflet originally has been developed as a part of the Geocluster module. During the development process, this part of Geocluster was generalized and published as the independent Leaflet GeoJSON module[13]. The visualization component of Geocluster therefore integrates with Leaflet GeoJSON and extends the JavaScript implementation of the Bounding Box strategy to integrate custom

[13] http://drupal.org/project/leaflet_geojson

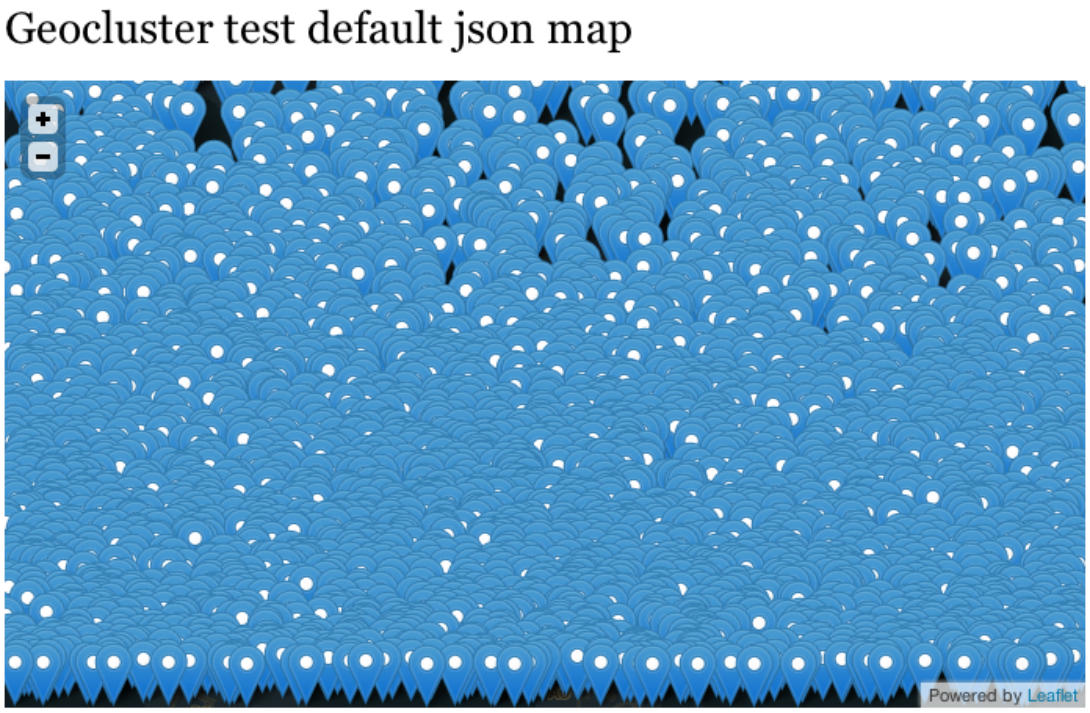


Figure 5.6: Unclustered Leaflet map.

markers and cluster interaction. The cluster visualization is based on a code snippet for numbered markers on github[14].

[14] https://gist.github.com/comp615/2288108

Chapter 6

Use cases

6.1 Demo Use Cases

A set of demonstration use cases has been created in order to test and evaluate the Geocluster implementation described in chapter **??**. The set consists of one non-clustering map and 3 maps based on the different clustering algorithms. The demo use cases were configured using various Drupal modules and exported into code using the Features module[1].

- **Geocluster Demo** show cases maps based on the two clustering algorithms provided by Geocluster module: PHP-based clustering and MySQL-based clustering and an additional map that doesn't use clustering at all. The article content type of a standard Drupal installation is extended by a Geofield-based place field for storing locations. For each map, a separate View is configured to provide a GeoJSON feed. A Leaflet map is then added on top of the feed by using the Leaflet GeoJSON module. Figure **??** illustrates a screenshot of a Geocluster Demo installation.
- **Geocluster Demo Solr** adds a show case of the Solr-based clustering algorithm. It provides a setup based on Views GeoJSON and Leaflet GeoJSON similar to the Geocluster Demo feature. In addition, a Search API Server and Index configuration is added for indexing and querying the data using Apache Solr.
- **Geocluster Demo Content** is a sub-module that automatically imports a set of demo content for testing the Geocluster Demo and Geocluster Demo Solr features.

[1] http://drupal.org/project/features

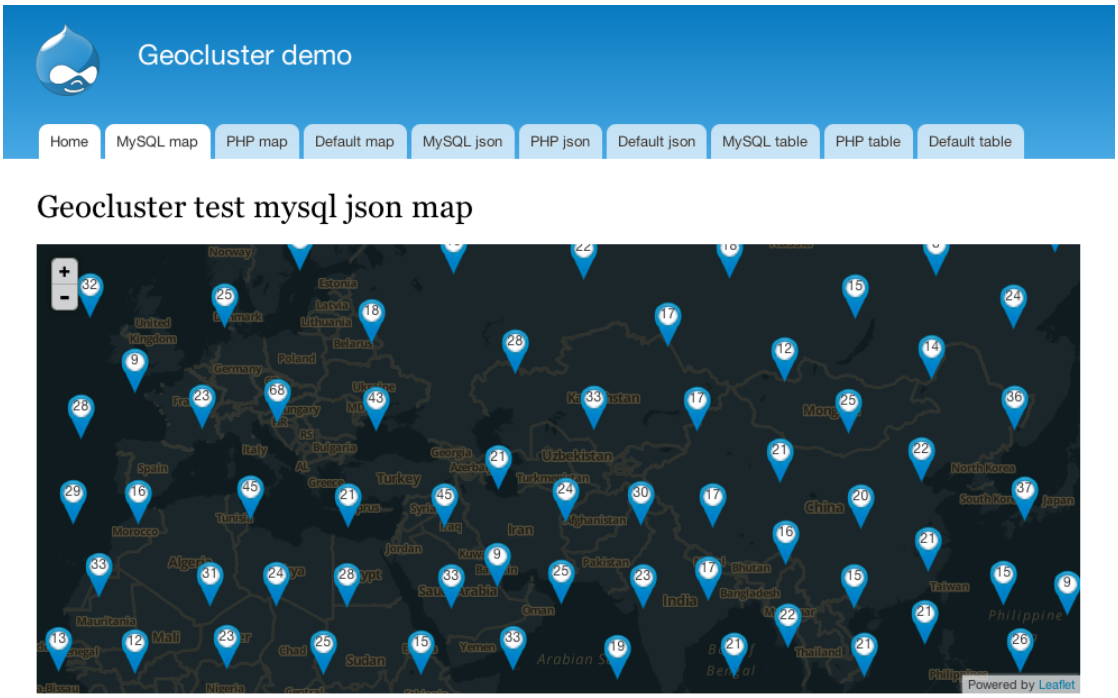


Figure 6.1: Screenshot of a Geocluster Demo installation. The active tab shows a map that uses MySQL-based clustering.

6.2 GeoRecruiter

A practical use case for server-side geo clustering has been implemented for the Recruiter job board solution which has been introduced in chapter **??**. It supports spatial search capabilities of the Recruiter distribution by visualizing a large amount of job offers on e-recruitment websites. GeoRecruiter allows to visualize several thousands of available jobs on an interactive map for large-scale e-recruitment websites. The Geocluster Solr module has been designed and used to provide the clustering capabilities needed by GeoRecruiter. The Solr-based aggregation integrates well with the architecture of the Recruiter distribution and is designed for scalability up to 1,000,000 indexed jobs as evaluated in chapter **??**. The prototype being discussed in the following chapter has been developed based on a copy of the Drupaljobs website[2].

Drupaljobs is provided by epiqo as a show case for the Recruiter distribution. Its base features allow to create and search for job offers by companies as well as resumes of registered applicants on the e-recruitment platform. Figure **??** depicts a screenshot of

[2] http://drupaljobs.epiqo.com

the heart of a Recruiter installation: the job search. The numbers on the figure indicate the main parts of such a page:

(1) A **search bar** above the content region.

(2) **Facetted filters** in the left sidebar.

(3) The **search results** as job teasers matching the search and filters.

For scalability reasons, the job search functionality of Recruiter is based on Apache Solr using the Search API module which have been introduced in chapter **??**. The concept of using facetted filters, allows the site visitor to narrow down the result set by applying filters based on properties of the result set. The screenshot from figure **??** displays filter facets based on *organization*, *fields of study* and *occupational fields*. Every filter item indicates the number of results to be expected when using this particular filter.

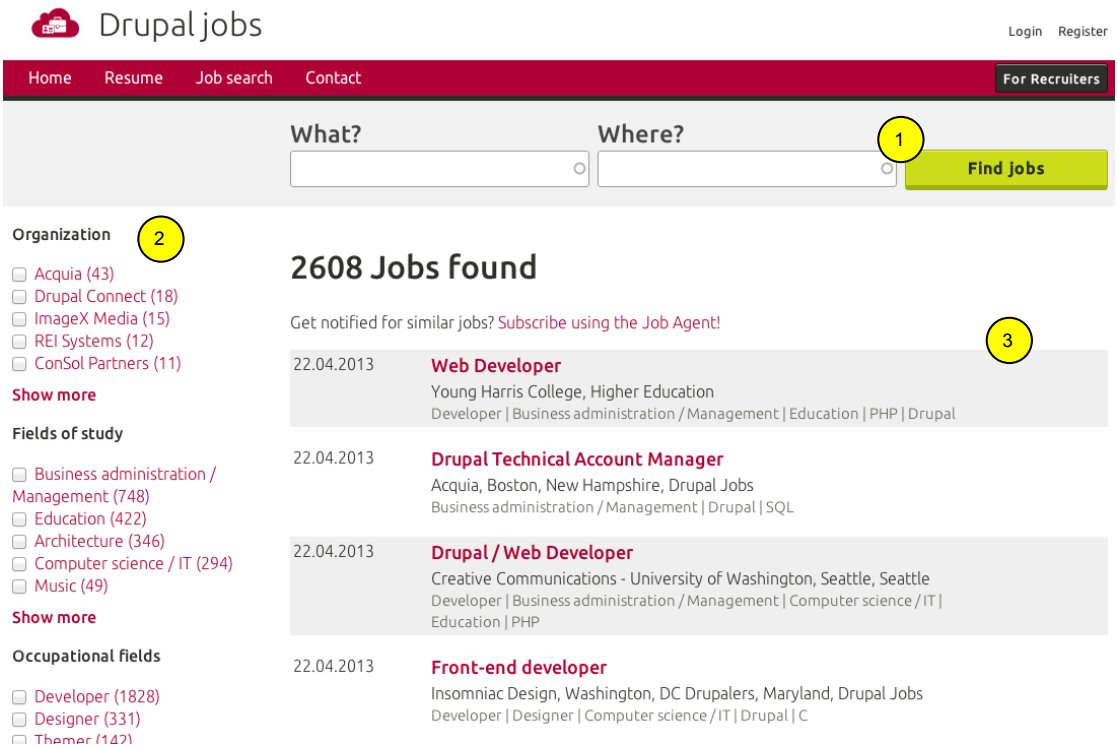


Figure 6.2: Screenshot of a job search on Drupaljobs including indicators: (1) search bar, (2) facetted filters and (3) search results.

The GeoRecruiter use case consists of several geo-related additions to the Recruiter distribution. As previously stated, the customizations have been prototyped using a Drupaljobs test installation.

- **Add geospatial data**: The data model for posting job offers of the Recruiter distribution has been extended to support the annotation of a geospatial location as the *place* property. In particular, a Geofield was added to the job node content types.
- **Import test data**: Two sets of real-world geospatial test data have been prepared for the Drupaljobs test site. A set of 10,000 world-wide cities was created based on a dataset from GeoNames.org[3]. Another set of 100,000 of U.S.-specific landmarks is based on a dataset from the U.S. Board on Geographic Names[4]. The kind of data isn't necessarily related to but will be mapped to job offers. This approach was taken due to the lack of a geospatially annotated datasets of job offers being available for testing purposes. Next, the test data was cleaned from errors and imported into the adapted Drupaljobs test installation. The import process was facilitated by using the Feeds module[5] which allows to import data into a Drupal site from external data sources like RSS feeds or in this particular case: CSV files.
- **Configure Geocluster Solr**: The server-side clustering component explained in **??** has been installed on the Drupaljobs test instance. The job search has been configured for clustering based on Apache Solr and Search API. Finally, a map visualizes the clustered job search results using Views GeoJSON and Leaflet. In order to enhance the representation of clusters and to experiment with interaction, the CSS styles of client-side clustering library Leaflet.markercluster have been adapted and extended with additional colors for large clusters.
- **Compare with client-side clustering**: In order to measure the effectiveness of the server-side clustering approach, a client-side clustering solution has been implemented for Drupaljobs as well. The client-side clustered map is based on a blog post by Ian Whitcomb of LevelTen [**?**]. For querying such a large dataset, he recommends circumnavigating the Views module and directly querying the database. The client-side clustering and visualization is again realized by the Leaflet.markercluster.

The resulting prototype allowed to experiment with the server-side clustering solution in a realistic environment, draw conclusions on effectiveness of clustering algorithm and

[3] http://download.geonames.org/export/dump/cities1000.zip
[4] http://geonames.usgs.gov/docs/stategaz/NationalFile_20130404.zip
[5] http://drupal.org/project/feeds

the visualization component being used. A visualization of a map within the Drupaljobs test installation is provided in figure **??**.

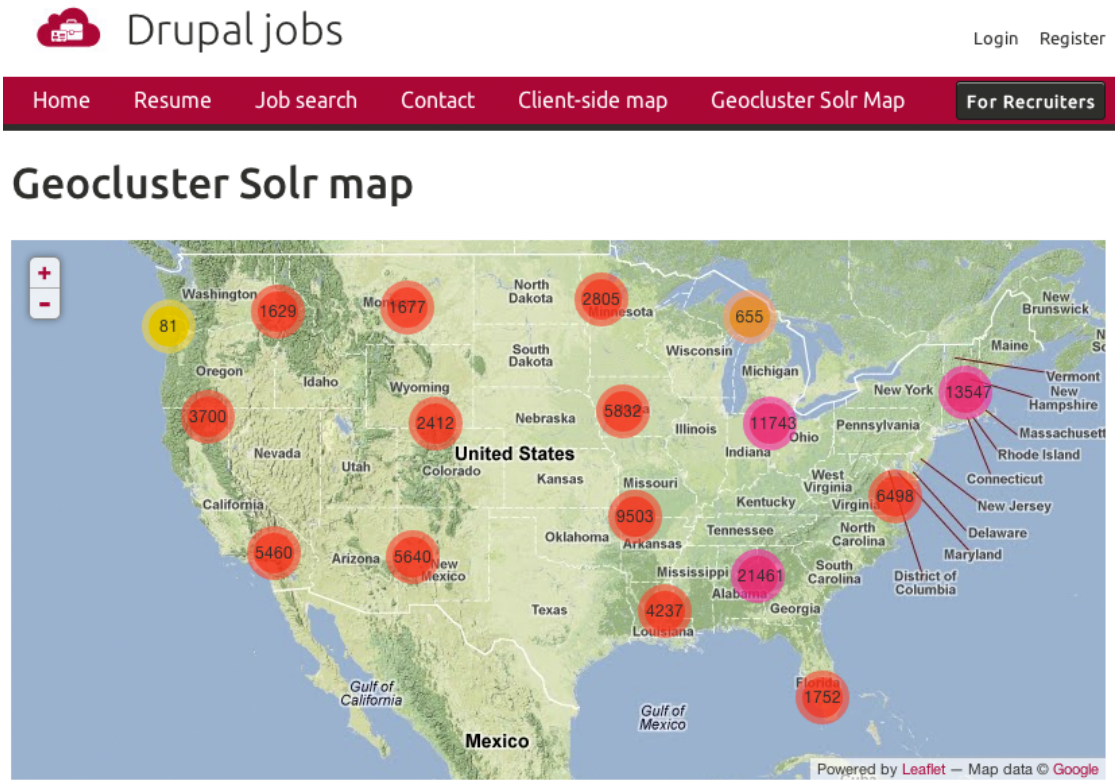


Figure 6.3: Screenshot of map that visualized job search results on a map using Solr-based clustering on a Drupaljobs test installation.

Besides the clustering functionality, GeoRecruiter will support location-based search. This allows the user to search for jobs within the surroundings of a desired region by applying a proximity filter. The Search API Location module is currently being refactored[6] in order to provide a solid foundation for such spatial queries using Solr and the Search API module suite.

[6] http://drupal.org/node/1798168

Chapter 7

Conclusions & Outlook

7.1 Performance evaluation

Performant real-time clustering is the main objective of this thesis as formulated in chapter **??**. As a requirement, the algorithm should cluster in real-time to support dynamic queries and cluster up to 1,000,000 items within less than one second.

The configuration of the demo use case implementation described in chapter **??** was used to do automated performance testing of the different clustering algorithms. A *Bash*[1] script was created to test the performance of the clustering algorithm based against an incrementing number of items.

The script exponentially increases items to test the clustering performance from a base 10 up to 1,000,000 items. Between every two steps of the exponential function, an intermediary step of the half of the two steps will be inserted. While the exponential mean value for example between 100 and 1000 items would be 316.2, this approach inserts 500 to improve readability of the results for humans. The resulting curve of items tested is visualized in figure **??**.

The *ab* command of the ApacheBench[2] is used to sequentially repeat the same requests and calculate a mean response time value. Hereby the script tries to circumvent variations caused by external factors as the server hardware and operating system.

The results of the performance benchmark have been extracted and aggregated into a chart as show in figure **??**. It is clearly shown that that the three implemented algorithms perform very differently. Each algorithm scales up to a certain number of items,

[1] http://en.wikipedia.org/wiki/Bash_(Unix_shell)
[2] http://en.wikipedia.org/wiki/ApacheBench

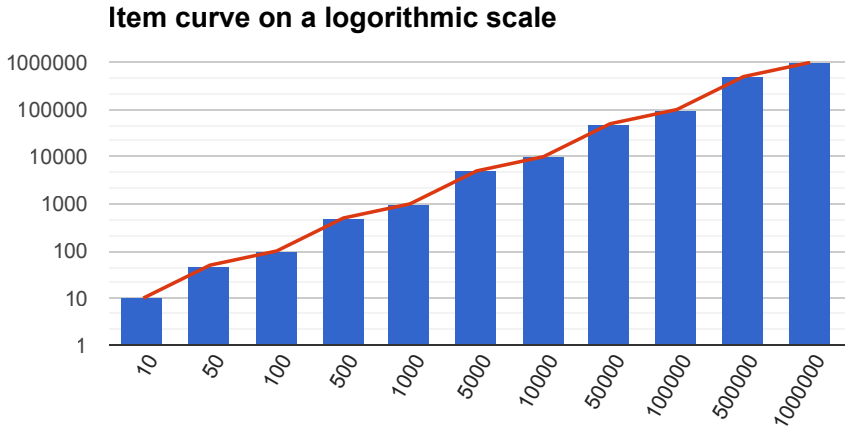


Figure 7.1: Item curve on a logorithmic scale.

while beyond this threshold, performance decreases significantly. The PHP-based clustering algorithm is very limited in such that requests for up to 1,000 clustered items can be completed within one second. The MySQL-based clustering approach scales much better but requests get slow beyond 100,000 items. The most performant algorithm implementation is the Solr-based one that server 1,000,000 items still in a reasonable amount of time.

A deeper analysis of the PHP-based algorithm clearly shows that the most time-consuming part of the algorithm is creating the clusters based on Geohash. As all unclustered items need to processed after executing the database query have to be processed, this part takes the most time. In an example based on a query with 9270 items, the entire roundtrip between client and server takes 26.71 seconds. Querying the items just took 100ms. With 24.32 seconds, the Geohash-based pre-clustering consumes the major part of the execution time. For the same amount of items, a request using the MySQL-based algorithm was completed within 194 ms. In this case, the query was completed after 80 ms and the while clustering process was finished 8 ms seconds later. The remainder of the processing time was consumed by Drupal performing non-clustering related processing before and at the end of the request. This example shows, how shifting the main clustering task into the database can increase performance for a certain range of number of items. As stated before, when approaching 100,000 items the MySQL-based algorithm is getting significantly slower, as the query itself takes longer.

Given the numbers, the performance criterion of this thesis could be fulfilled. While the PHP-based implementation isn't really usable, MySQL-based and Solr-based clustering can be used to create performant, interactive maps with Drupal for item sets up to at least 1,000,000 items. While we know that MySQL-based clustering only scales up to

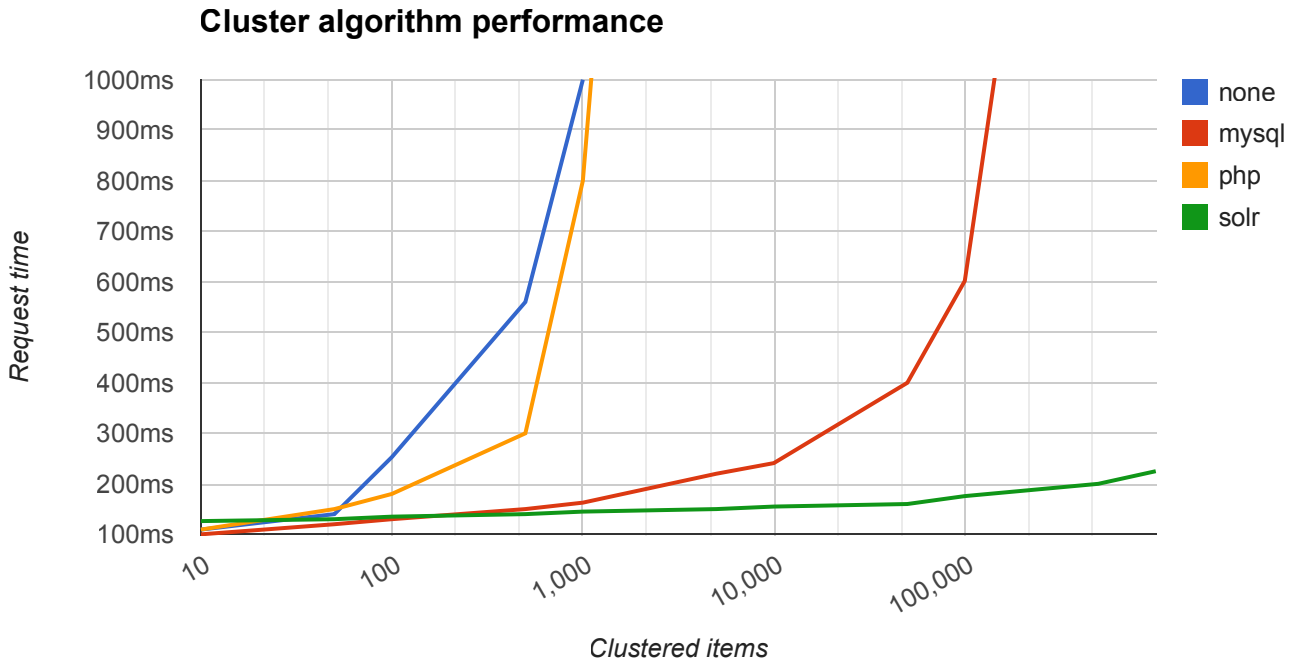


Figure 7.2: Geocluster performance in milliseconds per algorithm and number of items.

100,000 items, the threshold for Solr-based clustering wasn't determined as tests were limited up to 1,000,000 items. It is expected that there is room for improvements and optimizations for all of the algorithms.

Real-world scenario. In a second step, the performance of Geocluster has been evaluated based using a more realistic setup. The Drupaljobs test site based on Recruiter, explained in chapter **??**, was used as a test environment. A scenario for performance testing the map was created using the Selenium IDE[3] for the Firefox[4] web browser. Selenium IDE is an integrated development environment, that allows to record and play back tests. In combination with the Firebug[5] browser extension, the response times for the map interaction have been captured and evaluated.

The test scenario opens the website with a map that displays job offers using either the MySQL-based or the Solr-based clustering algorithm. The larger test data set of the Drupaljobs test site with 100,000 items within the U.S. was chosen. As the previous

[3] http://docs.seleniumhq.org/projects/ide/
[4] http://www.mozilla.org/en-US/firefox
[5] http://getfirebug.com/

performance test indicates, this amount of items can't be processed efficiently using the PHP-based implementation of the clustering algorithm, this is why PHP-based and Solr-based clustering were evaluated. In comparison to the previous performance test, this test aims at simulating standard user behavior in the browser. Thus, a Selenium script captured a sequence of user interaction on the map as zooming into particular areas of the map. In total, one sequence consists of the zooming into 4 different areas of the map on 5 zoom levels and zooming out again which sums up to 21 request per sequence. As the bounding box strategy of the Javascript mapping automatically applies a filter to the query for the current viewport, a different sub-set of the 100,000 items will get queried per request. The test sequence was repeated 5 times for each of the two clustering algorithm implementations and the results captured using the Firebug Net Panel log.

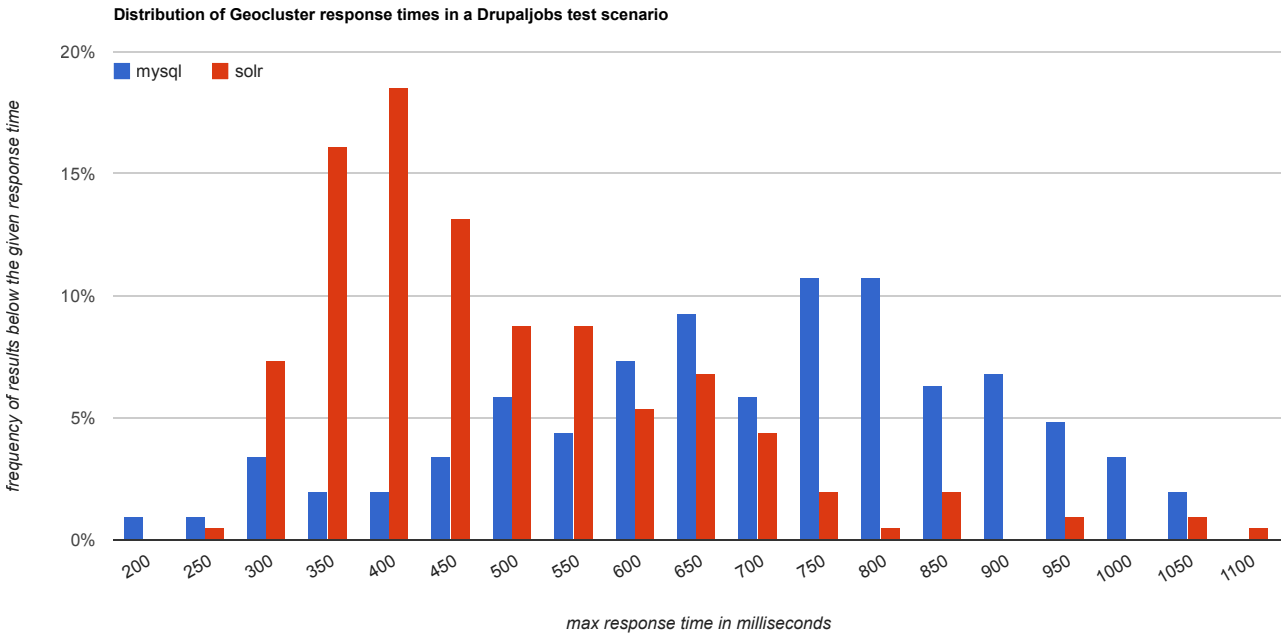


Figure 7.3: Distribution of Geocluster response times for MySQL- and Solr-based clustering in a Drupaljobs test scenario with 100,000 items.

The results of the real-world test scenario have been evaluated into a histogram that shows the distribution of response times and which is illustrated in figure **??**. The Solr-based clustering implementation shows a highest frequency of response times under 400 milliseconds, while the response times of the MySQL-based implementation have a highest frequency below 750 and 800 milliseconds. These real-world test results yield slower response times than the previous performance test where Solr-based response

times where constantly below 300 ms. This is might be caused by external factors: in the first case, a plain Drupal installation was being used and queried locally from the server, while the second test case was built on top of an existing Drupal site with a larger number of modules installed and queries where done from a local machine, adding network latency.

Another performance-related aspect is cachability of responses and results. Apache Solr and Drupal itself already incorporate various caching layers. Caching clustered results for the server-side clustering solution mainly depends on the parameters of the Bounding Box strategy. As currently, every minor change to the bounding box will issue a different request to the server, these can't be cached efficiently. A possible solution that has been discussed[6], is to define fixed steps for the bounding box in order to produce repeating requests that can be cached.

7.2 Visual evaluation

The second objective from chapter **??** defines the vague goal of supporting the user by compacting the amount of information that is visualized. To do so, two visualizations have been provided: first, the Client-side Geocluster Visualization component as visualized in figure **??** of chapter **??** and second, an alternative visualization similar to the Leaflet.markercluster library for the GeoRecruiter use case, see figure **??** in chapter **??**.

Both implementations fulfill most of the clutter reduction criteria from chapter **??**:

1. *Overlap* is avoided by visualizing a non-overlapping clustering of points, as created by geohash-based clustering algorithm of Geocluster.
2. *Spatial information* is expressed as an aggregate of latitude and longitude values for each cluster, approximating its centroid. One problem with the current implementations is, that clusters aren't "stable". In some experiments, changes to the bounding box will cause a change of cluster assignment. Intuitively, this leads to confusion of the user and should be investigated upon further.
3. The Leaflet Bounding Box strategy implementation allows to *localize* the view by panning and zooming and therefore reduce the display to a specific region.
4. *Scalability* is provided by the underlying clustering algorithm, as evaluated in the previous section.

[6] http://drupal.org/node/1868982

5. The configuration options of Geocluster, presented in chapter **??**, allow to *adjust* the minimum distance between clusters. Additional configuration options are provided by the Views integration of the Geocluster module. Still, the configuration options could be expanded for example to control visual parameters of clusters.

6. The criterion of *showing point/line attributes* is fulfilled only in a very limited way. Both implementations are currently restricted to displaying the number of items within a cluster as the only aggregate value. In order to support complex visualization techniques for multivariate data as discussed in chapter **??**, the clustering implementation needs to provide the required, aggregate values of the underlying data.

7. As explained in the discussion of the *discriminating points/lines* criterion, its definition seems unclear. Clustered items and individual points are visualized in a different way, which can be seen as a fulfillment. On the other hand, the visualization currently doesn't provide any means of inspecting clusters. Only a list of identifiers of the items within a cluster is provided. Based upon the identifiers, a popup could be used to visualize the items a in more detailed way.

8. With regards to *overlap density*, the number of items within clusters is indicated by both visualizations, but using different means. The Geocluster visualization doesn't make use of visual attributes like size or color, instead it creates a marker glyph that contains the number of items using textual representation. The GeoRecruiter prototype uses a color ramp that indicates low cluster densities from green and yellow to high densities indicated by tones of red and violet.

	category	shape	shows # of items within cluster	by shape size	by color or other	shows cluster area	shows extra cluster info	extra cluster info complexity
Geocluster default	example	marker	x	-	-	-	-	-
GeoRecruiter	example	circle	x	-	x	-	-	-

Figure 7.4: Evaluation of Geocluster visualization techniques for clusters on a map. Legend: 'x': yes, '~': possibly, '-': no.

Next, the evaluation of visualization techniques for clusters on map presented in **??** is applied to the two Geocluster implementations as illustrated in figure **??**. It reiterates some key aspects identified by the previous discussion of clutter reduction criteria.

Cluster sizes: Both visualizations show the number of items within a cluster, but only the GeoRecruiter example uses color and none of them encodes the size of a cluster into the shape size. As naturally, a cluster with more items can be visualized larger than smaller clusters, the algorithm could be improved for growing clusters by their

size. The bigger size of a cluster would therefore reduce the distance to its neighbor clusters, potentially merging additional neighbors into it. Andrew Betts describes a similar approach under the term *"Grid based viral growth argorithm"* [**?**].

For low zoom levels, the roundtrip to the server for fetching a separate clustered result on every bounding box change can be an overhead. Christopher Calid[7] proposes a way of "Progressively enhance server-side with client-side clustering"[8]. The intention is to switch from server-side clustering at higher zoom levels to client-side clustering for lower zoom levels.

7.3 Further evaluation

The third objective on integration and extensibility defined in chapter **??** has been fulfilled by the Geocluster module as it integrates with Views, Views GeoJSON and other Drupal mapping modules. In addition, the implementation of the clustering algorithm can be extended using plugins as explained in chapter **??**. Geocluster was also released under the GPL license as required by objective **??**. With regards to objective **??**, a demo use case has been implemented that show cases all the functionality needed. Also, the GeoRecruiter use case has been prototyped.

While most objectives have been reached, there are still many parts of the Geocluster implementation that can be improved upon. For example, the way that the Drupal mapping stack integrates with the Views module isn't designed for processing clustered results. The current implementation of Geocluster performs various workarounds in order to inject clustered results into the process. Especially for the Solr-based implementation this leads to code-duplication because in the standard case results are processed as arrays while in the other case, results need to be PHP objects. If possible, a cleaner way of integrating clustering with the related modules is desirable.

7.4 Conclusions

Writing this thesis and implementing Geocluster was basically a process of over one year. Before starting the thesis, I conducted a research project named AustroFeedr[9] on real-time processing technologies for aggregating, processing and visualizing data with Drupal. One main aspect of AustroFeedr was a Drupal-based visualization component using OpenLayers maps. After I had completed AustroFeedr by the end of 2011, I

[7] http://drupal.org/user/210499
[8] http://drupal.org/node/1914704
[9] http://www.austrofeedr.at/

researched Drupal and mapping related topics for writing this master thesis in Software Engineering & Internet Computing at Technical University Vienna.

The topic server-side clustering for Drupal was decided upon thanks to recommendation by Théodore Biadala[10], an active JavaScript and maps contributor in the Drupal community who I met at the Frontend United conference in Amsterdam, April 20-22. After doing some initial research and prototyping, I organized a mapping sprint at Drupal Developer Days Barclona[11]. This is where Nick Veenhof[12], active Apache Solr contributor within the Drupal community, came up with the idea of researching Geohash for realizing an efficient clustering algorithm.

It took until September 2012, when I implemented a first prototype of the PHP-based clustering algorithm and figured out basic integration needs for to make the clustering task work with Drupal. From then, several iterations and alpha releases of Geocluster led to completing MySQL and Solr-based clustering by the end of 2012. As by finishing this thesis in March 2013, performance tests have been concluded for Geocluster and a first beta release has been published.

There has already been some positive feedback from people interested in using Geocluster as a drop-in solution to create scalable maps using server-side clustering. On the other hand, I have to admit that integrating clustering into a complex stack such as the Drupal mapping stack has its advantages and disadvantages. Geocluster does a decent job at clustering data server-side, but the tight integration into the Drupal stack also comes at the cost of overhead and complex integration code. For a person that has the expertise in writing code, it might make sense to create a custom server-side clustering solution for a specific purpose without depending on a number of separate modules. Still, the generic approach has the benefit of others potentially being able to use the Geocluster module. Writing this thesis was both challenging and fun. Drupal is a steadily growing team of contributors of Free and Open Source software. Being able to share my works with such a great community has been a rewarding experience and a solid source of motivation.

7.5 Future work

Direct community feedback and indirect indicators like the project usage statistics will show if and how the Geocluster module is used by others. As stated before, there are many implementation details that can be enhanced. At epiqo, we are planning to in-

[10] http://drupal.org/user/598310
[11] http://groups.drupal.org/node/234168
[12] http://drupal.org/user/122682

corporate Geocluster for location-based searches of large-scale job portals based on the Recruiter distribution as stated in chapter **??**.

Appendix A

Acronyms

AJAX Asynchronous JavaScript + XML

API Application Programming Interface

BBOX Bounding Box

CMS Content Management System

DBSCAN Density-based spatial clustering of applications with noise

GNU GNU's Not Unix

GeoJSON JSON for geographic data structures

GIS Geographic Information System

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IDE Integrated Development Environment

IT Information Technology

JSON JavaScript Object Notation

OASIS Organization for the Advancement of Structured Information Standards

PHP PHP: Hypertext Preprocessor

REST Representational State Transfer

RSS Really Simple Syndication

SOM Self-organizing maps

STING Statistical Information Grid

TMS Tile Map Service

UDDI Universal Description, Discovery and Integration

UI User Interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

W3C World Wide Web Consortium

WMS Web Map Service

Appendix B

Index

List of Figures

List of Tables

Listings