FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Geospatial Information Management in Web-Based CMS Systems

Theory, techniques, specifications and data management

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Magister rer.soc.oec

im Rahmen des Studiums

Magisterstudium Informatikmanagement

eingereicht von

Jeremy Chinquist
Matrikelnummer 0526895

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gerald Futschek
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.rer.soc.oec. Amin Anjomshoaa

Wien, 18.08.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Geospatial Information Management in Web-Based CMS Systems

Theory, techniques, specifications and data management

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Magister rer.soc.oec

in

Master Programme Computer Science Management

by

Jeremy Chinquist
Registration Number 0526895

presented to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gerald Futschek
Assistance: Univ.Ass. Dipl.-Ing. Dr.rer.soc.oec. Amin Anjomshoaa

Vienna, 18.08.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Jeremy Chinquist
Raxstrasse 32 / 107, 1100 Vienna, Austria

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.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

Dedicated to my wife Marion and our parents James, Caroline, Alois and Martha.

Kurzfassung

Diese Masterarbeit beschäftigt sich mit der Verwaltung von Geoinformationen in quelloffenen webbasierten Content Management Systemen (CMS). Geoinformationen können von CMS in mehreren der Technologie-Layer, auf der Datenbank-Ebene, im Dateisystem als auch auf der Programmierebene verwaltet werden.

Häufige Anwendungsfälle schließen das Speichern von Ortsangaben oder Gebietsangaben bis zur Speicherung von Objekten ein als auch die Evaluierung deren Beziehung zueinander. Beispiele wären der Abstand zwischen zwei Objekt-Sets oder ob ein Datenobjekt innerhalb der Grenzen eines anderen Datenobjekts liegt. Abhängig vom Anwendungsfall können die Objektdaten in verschiedenen Formaten exportiert werden, zum Beispiel als GeoRSS Stream, oder als GeoJSON, KML, GML, XML, GPX und RDFa Datenbestände zur Anwendung in Systemen Dritter. Das Zusammenspiel mit externen Systemen und Speichern von Geoinformationen ist eine zentrale Aufgabe der Verwaltung von Geoinformationen.

Am Anfang dieser Arbeit werden typische Anwendungsfälle für webbasierte Content Management Systeme identifiziert und evaluiert, um folgende Forschungsfragen zu beantworten: Wie sollen Geoinformationen von CMS strukturiert und verwaltet werden, um die Daten möglichst effizient und flexibel verwalten und darstellen zu können? Welche Datenbestandteile, falls überhaupt, sollen (lokal vom CMS) gespeichert werden? Welche externen Quellen an Geoinformationen sind verfügbar und wie können diese Quellen effizient genutzt oder in das CMS importiert werden?

Drupal 7 und Wordpress 3.9.1, zwei der populärsten quelloffenen webbasierten Content Management Systeme, werden evaluiert. Es wurden Prototypen der Content Management Systeme geschaffen und installiert, und mehrere Plugins (Wordpress) und Module (Drupal) evaluiert. Des Weiteren wurde Forschung bezüglich Technologie-Stacks betrieben.

Wie gezeigt wird, verwenden beide Content Management Systeme jenen Datentyp der Geoinformation nicht, welcher in relationalen Datenbanksystemen wie MySQL und PostGIS verfügbar ist. Der Datentyp ist den numerischen und Stringdatentypen ähnlich, ermöglicht aber komplexere geoinformationsbezogene Abfragen wie ST_Contains, ST_Within und MBRContains und MBRWithin. Effizienztests wurden erstellt, um zu evaluieren, ob die Content Management Systeme vom Datentyp profitieren.

Die Effizienztests zeigen, dass der Geoinformationsdatentyp die Ergebnisse für ST_Contains und ST_Within schneller liefert als nicht-Geoinformationsdatentypen, aber nicht für Rechtecksfunktionen wie z.B. MBRContains. Allerdings sind ST_Within und ST_Contains sehr wichtige Funktionen und deshalb ist es empfehlenswert auf Geoinformationsdatentypen umzusteigen.

Abstract

This thesis evaluates the management of geospatial information in Web-based Open Source CMS. Geospatial data can be managed by the CMS in several of the technology layers, at the database level, on the file system level as well as in the programming level.

Common use-cases include saving location data or area data to objects and evaluating their relationship to one another. Examples include the distance between two data objects or whether one data object is within the bounds of a different data object. Depending on the use-case, the object data can be exported in several different formats, such as a GeoRSS stream, or as GeoJSON, KML, GML, XML, GPX and RDFa data files for use in third party systems. Interacting with external systems and geospatial repositories is a central part of geospatial data management.

The following questions were addressed: How should geo data be structured by (web-based) CMSs so that the system, and the administrator, can efficiently and flexibly manage the geo data and display it in context? What data, if any, should be stored, locally within the CMS? What external sources of geospatial data are available and how can these sources be efficiently used or imported into the CMS?

Two popular Open Source Web-based CMSs were evaluated, Drupal 7 and Wordpress 3.9. We created prototypes of the CMSs and installed and evaluated several plugins (Wordpress) and modules (Drupal) as well as conducted research on the technology stacks.

As is shown, neither CMSs take advantage of the Geospatial data type available in relational database systems like MySQL and PostGIS. The data type is similar to String and Numeric data types, but allows for complex geospatial equations such as ST_Contains, ST_Within and bounding box functions to be made on the data in the columns. Timing tests were developed to evaluate whether the CMSs should take advantage of the geospatial columns.

The timing tests show that geospatial columns are faster for functions such as ST_Contains and ST_Within, but not for the bounding box functions like MBRWithin. However, ST_Within and ST_Contains are very important functions and therefore it is still advisable to switch to geospatial columns.

Contents

Contents **ix**

1 Introduction **1**
1.1 Motivation 1
1.2 Research questions 4
1.3 Outline and methodology 5

2 Mathematics of Geospatial Data **7**
2.1 Geographic coordinate system 7
2.2 Computing distances 8
2.3 Computing the area of regions 10

3 Use-cases of geospatial data management in Web-based Open Source Content Management Systems **11**
3.1 Use-cases 11
3.2 Display the raw geospatial data on a web page 13
3.3 Display a geospatial point on a static map 13
3.4 Display a geospatial feature on an interactive map 14
3.5 Print shapefiles 15
3.6 Print a Keyhole Markup Language (KML) or Geography Markup Language (GML) file 16
3.7 Print GPX data 17
3.8 Print GeoRSS data 18
3.9 Print GeoJSON data 19
3.10 Print Geo Microformat, hCard Microformat or Schema.org markup 20
3.11 Print geo meta tags to a webpage 22
3.12 Geocode and reverse-geocode an address 22
3.13 Obtain distance and directions between two points 24
3.14 Map clustering and geohashing 24
3.15 Calculate additional properties of geospatial data 25
3.16 Calculate relationships of two geospatial features 25
3.17 Import geospatial data from and export to external repositories 26

4 State of the Art **33**
4.1 Wordpress . . . 33
4.2 The database layer and the OpenGIS Geometry Model . . . 34
4.3 The State of the Art of Drupal . . . 38

5 Methodology and Testing **49**
5.1 Methodology of the Tests . . . 49
5.2 Hypotheses and expected results . . . 55
5.3 Results . . . 55

6 Discussion of Results, Conclusions and Outlook **65**
6.1 ST_Contains & ST_Within . . . 65
6.2 MBRContains . . . 65
6.3 AREA . . . 66
6.4 Distance from a point using the Haversine Formula . . . 66
6.5 Conclusion of research questions . . . 66

A Appendix A: Index **69**
A.1 Glossary . . . 69
A.2 List of Acronyms . . . 70

List of Figures **71**

List of Tables **73**

B Appendix B: Timing Tests **75**
B.1 Synchronisation of GeoField Table for Geospatial Column Values . . . 75
B.2 Synchronisation of GeoField Table for Radians, Sine and Cosine . . . 77
B.3 Timing Test ST_Within and ST_Contains . . . 78
B.4 Timing Test MBRContains . . . 81
B.5 Haversine Great Circle Distance Formula Timing Tests . . . 83

Bibliography **87**

CHAPTER 1

Introduction

> "One of the great science paradign shifts in our age is that everything is now seen as related to everything else, whereas the emphasis in the previous two centuries was on studying phenomena in isolation. Consequently, in the geosciences as in other sciences, studies are more frequently interdisciplinary. This trend is consistent with the increased sharing of data, information and knowledge that characterizes our increasingly networked world." - Open Geospatial Consortium

1.1 Motivation

Website designers and CMS programmers are increasingly being forced to deal with the management of geospatial information [1] [2]. Whether it is to simply display one location on a static map for a small website, or to provide interactive maps that the readers and CMS users will use as part of the data management tools, the administrator must choose from a wide range of tools. To diplay that information in a context that can be used by mobile devices such as smart phones, tablet computers, laptops, medical devices, et cetera, or by stationary devices such as desktop computers and televisions or playstations calls for increasingly complex systems.

Geo services such as Google's Geocoder service, database repositories like Linked Data or GeoNames and the newly emerging mapping technologies such as Leaflet, Open Layers 2.11 and Google Maps API v3 are forcing client programs, CMSs and developers to adapt rapidly. First generation systems, where geospatial data was stored exclusively on the local machine in files or a database as floating point coordinates, have given way to third generation systems. Systems that geocode address information or administer more complex geospatial data than point data, such as linestrings or even boundaries. Furthermore, the spatial data is often stored in a decentralized fashion in which the information is no longer stored locally or multiple applications interact with the data.

[1] Google Maps Usage Statistics, http://trends.builtwith.com/mapping/Google-Maps
[2] Usage statistics for Geolocation Field, http://drupal.org/project/usage/geolocation

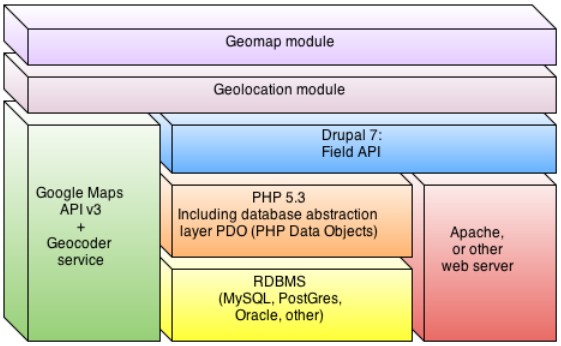

Figure 1.1: The technology stack of the Geolocation and Geomap modules on top of Drupal and its required technologies

Today's leading Web-based Open Source CMSs make use of a web server, often Apache or Tomcat, a programming language to serve dynamic pages, such as PHP or Java, and a (relational) database software and can interact with mapping technologies like Google Maps to present geospatial data on a visual map.

In this paper, two of the most widely adopted Web-based Open Source Content Management Systems that are mentioned are Wordpress and Drupal, but there are several others that are worth mentioning, like Zend Framework, Joomla and non-PHP based CMSs such as Ruby and Plone. Written both in the PHP programming language, Drupal and Wordpress' most common technology stack is Apache, PHP and MySql running on Linux. See 1.1 for an example stack.

Wordpress is a lighter weight Open Source Web-based CMS that primarily supports blogging use-cases such as saving and serving up static pages and comments. Wordpress' philosophical approach includes striving for simplicity and being an out of the box solution [3]. This includes offering a set of decisions for how blog posts should be managed and presenting the authors and editors with fewer options. Therefore Wordpress' architectural decisions, even for geospatial information management, is often to write data in the wp_options table as a Post ID, Key, Value tuple, which are managed through plugins.

Drupal is a more complex CMS that is designed to handle more use-cases than blogging and does not set constraints on administrative options. The CMS has an elaborate API toolset, including the powerful FieldAPI system, where new field types can be defined through contributed modules and instantiated through the administrative user interface (UI). Modules are responsible for validation behaviour and handling database storage limitations. Drupal is not just concerned with presenting web-pages in the form of posts, but can be used as a geospatial database service or a custom API. The use-cases are much broader in scope. Several use-cases are also covered in the book "Mapping with Drupal" [Pal07].

CMSs inevitably must manage geospatial data, whether it be simple address data or storing areas and making complex queries on that data and saving it to the relational database management system (abbreviated RDBMS). According to Ray et. al., the amount of business data

[3] Wordpress Philosophy

stored in existing databases with spatial attributes was estimated at 80% a decade ago [Rax11] and estimates that figure to be higher today.

Geospatial data is any data that represents a geographic feature. Relational databases such as MySQL, MS SQL, Oracle, PostgreSQL, and the like are considered spatial databases. Spatial databases are able to store spatial data in the same manner as numerical data (floating point, integer, etc.) and string data (variable character data, blobs, etc.). In MySQL these columns that store spatial data are called geometry columns. Unfortunately, very little is done in CMSs to make use of geometry columns.

There are a number of services and methods for storing geospatial data. Choosing the correct sources and services that will continue to serve the CMSs over a long-term basis is often more challenging for the developer than actually configuring the Web-based CMSs and saving a few values to the database. The operations and use-cases that the CMS has to fulfill can mean either overkill on the data or not being able to fulfill the use-cases.

For example, the vCard specification for address information has been around since 1995 and is still supported by e-mail client programs such as Microsoft Outlook. A successor of the vCard is the microformat specification hCard that is used for websites. When viewing an address book entry on a website, should the administrator choose to print the hCard, the vCard or both to the page?

A higher level example is Geoclustering points on a map. This is a very expensive computation that needs to be done on the fly depending on the map zoom level and the current viewing area, known as a viewport. As has been shown in *Server-side clustering for mapping in Drupal based on Geohash, when using the server locally, performance degrades exponentially when using PHP to compute the clustering at less than 100 data points. If MySQL is used then performance degrades starting at 100,000 data points. Using an external service allows for 1,000,000 data points to be efficiently clustered. [Dab13]*

Context is also a difficult issue for geospatial data. There are technologies that will handle the display for the website administrator depending on the context of the reader. Context is defined both as "... the set of information that could be used to define and interpret a situation in which agents interact ... " and in "... the context-aware applications community, the context is composed of a set of information for characterizing the situation in which humans interact with applications and the immediate environment." [Bis]. Applying the correct context for geospatial data is not a simple task. Computer systems are binary systems where the data is either marked as "display" or "don't display", thus the information could be relevant to the reader for one data set, but lack any meaning in the next data set because of a difference in context.

In the worst case, improper context can lead to confusion. Consider a geospatial database of city names. The CMS user is able to choose a city for a location from that central database and intends to choose "Neunkichen" in Germany. If the CMS user is not in that country - for instance in Austria - the database may return "Neunkirchen" in Austria without the CMS user knowing that the data set is no longer accurate.

Additionally, web-developers are continually confronted with the request to display relevant geospatial data that is "in the area" of the current data set that possesses a location. Another variant is to display data sets that are "in the area" of the current location of the reader. Both use-cases are expensive for the server.

Other requests involve making the telephone number highlightable by the skype plug-in relevant to the country code, as well as the driving directions be made accessible if the reader requests it. Developers are overwhelmed by the requests, how does the administrator know which solutions will be good long-term solutions? Once the system is in place, if the data is not structured correctly, the simple algorithms of determining the distance between two locations can bring down a website because the server must compute the distance on every page request. [Rub06]

This paper is written for administrators, maintainers and designers of websites and web-based CMS systems who require geospatial solutions for their systems. Current data management techniques will be evaluated on a per use-case basis and it will go so far as to propose changes to optimize geospatial data management techniques currently available in those systems.

1.2 Research questions

This paper is concerned with the use-cases and efficient management of geospatial data and as such, the following questions are investigated.

- *How should geo data be structured by (web-based) CMSs so that the system, and the administrator, can efficiently and flexibly manage the geo data and display it in context?*
  - *What data, if any, should be stored (locally)?*
  - *In what ways should the geo data be structured?*
- *What external sources of geo data are currently available and how can these sources be efficiently used or imported into the CMS?*

How should geospatial data be structured by web-based CMSs so that the system can efficiently and flexibly manage the geospatial data and display it in context? To answer the first group of questions, the state of the art for both Drupal and Wordpress will be evaluated and compared to the common model for storing geospatial data, known as the OpenGIS Geometry Model.

The decision of structure is often dictated by the use-cases that need to be addressed by the CMS system. For example, the system does not need to save human readable address string data when only the latitude and longitude coordinates must be saved to display on a map. However, if one needs to do math that involves distance calculations, then often the sine and cosine of the latitude and longitude need to be calculated. Automatically saving those values can have performance benefits. Lastly, the altitude information is often irrelevant for displaying points on a 2 dimensional map. Only when the use-case involves displaying data on a 3D map such as Google Earth does the altitude play a larger role in the data. The use-cases will also be outlined and discussed in depth.

Several use-cases involve interacting with data repositories that are non-local. There are, in fact, several available, often free, geospatial database repositories and the trend is increasing. What methods are available to the Open Source CMS to interact with non-local repositories? How can data be imported and exported?

1.3 Outline and methodology

Before evaluating the use-cases themselves, a short introduction into the basic mathemtics behind geospatial data management will illustrate the efficiency and methods available to manage geospatial data.

The use-cases for geospatial data management as pertains to Web-based CMSs will be identified. What goals and purpose the data management tools aim to fulfill and how they currently fulfill those goals will be explored.

Then we will evaluate the state-of-the-art for geospatial data management in Wordpress and Drupal, two leading Web-based CMSs. Neither system possesses geospatial data management out-of-the-box, but they do, however, possess a variety of plug-ins (in the case of Wordpress) and modules (in Drupal) that extend the core functionality of the systems to manage geospatial data and fulfill specific use-cases. These modules and plug-ins make use of the core CMSs data management technologies, especially their APIs, to optimize performance of the systems.

Even though these systems fulfill many of the use-cases that are later outlined, the need for more efficient and flexible systems exist [Rax11]. In the Improvements chapter it will be illustrated that the relational database, which both Wordpress and Drupal depend upon, can manage geospatial data as a field type native to the database and that both CMSs can benefit if they make use of the database technology. Current plug-in and module versions do not take advantage of the geospatial data types in the database.

A geospatial database column is a column data type that can be added to any table in the database. As opposed to specifying a column as numeric, i.e. int or double, or as string, i.e. varchar or text, it can be specified as a geospatial type. Geospatial data types include the generic GEOMETRYCOLLECTION or more specific collections of geometries such as CURVE, LINESTRING, POINT, POLYGON and SURFACE. Each column data type is optimized for storing spatial data of that type. For more on the geometries and mysql, see Chapter 4. Figure 1.2 shows the data types available to a MySQL 5.7 database.

A series of installations of both Wordpress and Drupal were created to model and investigate the use-cases of geospatial data management. Research was conducted on these use-cases, including exchanging geospatial data between the CMS and external repositories.

Finally, we show that by using geospatial columns in the database, the workload can be shifted away from the php level and make the query speed more efficient. We do this by creating timing tests to show that using a geospatial database column increases the efficiency of the queries.

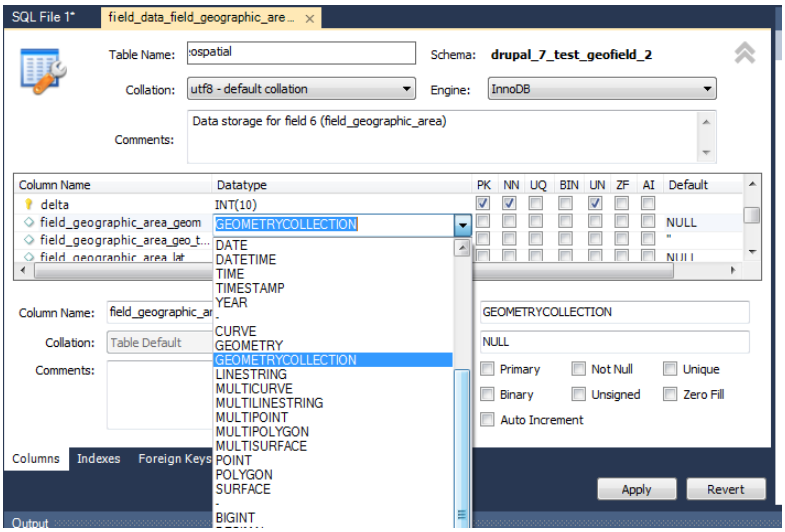

Figure 1.2: MySQL geospatial column data types

CHAPTER 2

Mathematics of Geospatial Data

According to MySQL and the Open Geospatial Consortium [Her10]

> *A geographic feature is anything in the world that has a location. A feature can be:*
>
> 1. *An entity. For example, a mountain, a pond, a city.*
> 2. *A space. For example, town district, the tropics.*
> 3. *A definable location. For example, a crossroad, as a particular place where two streets intersect.*

The Open Geospatial Consortium (OGC) more specifically states that geospatial data is synonymous with geospatial features and simple geospatial feature collections. A simple feature is defined as possessing both spatial and non-spatial attributes where spatial attributes are geometry valued. Simple features are based on a 2-D geometry, also called Euclidean Geometry, or flat-earth.

2.1 Geographic coordinate system

CMSs widely uses the geographic coordinate system using latitude and longitude. Lines of latitude are the imaginary rings around the earth parallel to the earth's equator (defined as 0 degrees latitude) and having a value between +90 degrees latitude (the physical north pole) and -90 degrees latitude (the physical south pole). Lines of longitude, or Meridians, the imaginary lines drawn along the earth's diameter connecting the earth's physical north and south poles with the Prime Meridian defined as the line of longitude that runs through the Royal Observatory in Greenwich, England and is given the value of 0 degrees longitude. Lines of longitude to the east are positive and lines of longitude to the west are negative. +/-180 degrees longitude are the same line, directly opposite the Prime Meridian. All locations on the earth can be defined as points on the latitude and longitude coordinate system.

Typically a map or GPS enabled device has an accuracy of 7.8 meters [-08]. A single degree of latitude is approximately 111 kilometers of physical distance. 7.8 meters is approximately 0,00007027 of a degree of latitude. Computer systems rarely have to save more than 6 decimal places in order to accurately store latitude and longitude data. Anything higher than 6 decimal places and the GPS device or the mapping technology is not accurate enough to display the data. Geocoder services such as Google's Maps API v3 Geocoder Service has an accuracy of 6 or 7 decimal places. To reduce complexity, examples in this paper will round to 4 decimal places.

Servers tend to have adequate storage space however, so the general practice in CMSs is to record coordinate data as decimal numbers. Default for most web servers is 10 to 14 decimal place precision. Future improvements in mapping GPS technology would then be supported by the software.

Degrees of latitude and longitude are often displayed and saved in databases as decimal or floating point numbers. The more human-readable method is to display the degrees-minutes-seconds, where there are 60 minutes to a degree and 60 seconds to a minute.

Unfortunately there are several variations on the basic geospatial coordinate systems. Each system is called a Spatial Reference System (SRS). Spatial Reference Identifiers (SRID's) are defined to specify in which Spatial Reference System a particular set of data is saved so that conversions can be made to go between the different SRS's. The most widely used spatial reference system is WGS 84 lat lon, which has the SRID EPSG:4326, because it covers the entire globe. cite:sync-postgisdrupalmodule Open Source GIS communities will often refer to the SRID 900913 as the Spherical Mercator projection which treats the earth as a perfect sphere rather than as an ellipsoid. The Spherical Mercator project is the projection used by Google Maps, Microsoft Virtual Earth, Yahoo Maps, as well as others. Different projections have different distortions which also vary depending on which areas of the world the projection focuses on.

Database data can be set to a specific SRID by using a functional index that indicates all data in a single spatial data column in a table has the same SRID. Common database engines also possess methods that allow it to translate data between SRID's, for example "ST_Transform" in a postGIS database.

2.2 Computing distances

On a 2-D map, there are several distance computation techniques, the most relevant for Web-based CMSs and their use-cases are the Euclidean distance and the Haversine formula methods.

Other methods worth mentioning because they are relevant to geoclustering algorithms are the Manhattan distance formula, which computes the average of delta x and delta y as the distance between two points, and the Chebychev distance formula, which assumes that distance is the greater value of either delta x and delta y. [Mee06]

Euclidean distance works best on 2-D map environments where the distance is measured as a direct line on the Cartesian coordinate system using the deltas of x and y. The greater the distance, the larger the distortion between the actual distance. The method however is very fast to compute and therefore is used by many servers where accuracy is not important and distance is not large.

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

Great-cirlce distance and the Haversine formula

Creating a direct line between two locations on a sphere is a chord and that line does not follow the surface of the earth, and instead travels through the middle. Therefore the distance distortion increases as distance increases.

One fairly simple method is to reduce the earth to a perfect sphere. The over-simplification of the earth's bulges, trenches, mountains and valleys allows for computer models that are less complex and therefore less server intensive.

The Haversine formula is one such formula that can be used to model the distance on a perfect circle.. [Rub06]

Suppose that latitude and longitude are given in decimal format. Determining the surface distance on a sphere is performed in radians, the first step is to translate the latitude and longitude ((LatA, LonA) and (LatB, LonB) respectively) coordinates to radians. Should the latitude and longitude not be in decimal format, but expressed in degrees, minutes & seconds, the server or client computer must first translate the values.

$$angleradians = \frac{\pi * angledegrees}{180}$$

The radius of the Earth is dynamic, with land altitude, mountains and sea levels having effects on the radius at any given point. To assume a perfect sphere, a value must be agreed upon. When using the Spatial Reference System WSG84, the Equatorial radius is defined as 6,378,137 meters and is therefore often used as the radius of the earth in computations. The Drupal GeoPHP module uses 6,378,137 meters. The radius at the poles however is 6,356,752 meters but is not used as often. Depending on the location of the points where the distance is being computed, a different value for the radius of the earth may bring a greater accuracy.

*The great circle distance (*d) between two points with coordinates (LatA, LonA) and (LatB, LonB) expressed in decimal format and assuming the radius of earh (R) can be calculated as: [Wil12]

$$d = 2R * \arcsin(\sqrt{(\sin^2(\frac{(LatA-LatB)*\pi}{180*2})) + (\cos(\frac{LatA*\pi}{180}) * \cos(\frac{LatB*\pi}{180}) * (\sin^2(\frac{LonA-LonB)*\pi}{180*2})))}$$

If LatA and LatB are saved in radians, the formula is reduced to:

$$d = 2R * \arcsin(\sqrt{\sin^2(\frac{(LatA-LatB)}{2}) + \cos(LatA) * \cos(LatB) * \sin^2(\frac{LonA-LonB}{2}))}$$

The accuracy of the calculations is also dependent on the accuracy of the input. Services such as Google's Geocoder Service will return latitude and longitude values to at least 6 decimal

places and as stated previously, 6 decimal places is the accuracy that most GPS enabled devices are capable of handling.

Some computations on the data can be stored in a database as well to reduce the number of calculations in the functions. Determining the angle in radians from the angle in degrees will take two computations per lookup, and, in each of the formulas, the sine and cosine of a single coordinate of a single point is determined multiple times. If the server has adequate storage space for an additional 3 floating point columns per data set, then it should save these values to the database.

2.3 Computing the area of regions

Regions are expressed as polygons in computer science and polygons are discussed further in the database layer and the OpenGIS Geometry Model.

Polygon areas in computer programs are most often computed on a 2-D surface because the area is computed using the endpoints of each line segment on the polygon. Computing the surface area on the great circle requires much more complex calculations. To compute the area of a closed and simple polygon on a 2D surface, use the following equation:

$$area = \left| \frac{(Lon_1 * Lat_2) - (Lat_1 * Lon_2) + (Lon_2 * Lat_3) - (Lat_2 * Lon_3) + ... + (Lon_n * Lat_1) - (Lat_n * Lon_1)}{2} \right|$$

Just like the Euclidean distance formula, the closer the points on the polygon, the more accurate the area measurement. Also, the larger the surface area of the region, the more will be lost when mapping the data to the flat surface.

CHAPTER 3

Use-cases of geospatial data management in Web-based Open Source Content Management Systems

3.1 Use-cases

Whether it be done in Wordpress for blogging or in Drupal or another Web-Based CMS for article publishing and advanced content management, the following use-cases are the most common:

1. Create, update and delete (CRUD) geospatial data locally in the CMS
2. Display the raw geospatial data
3. Display geospatial features on a static map
4. Display HTML geolocation attribute [1]
5. Display geospatial features on an interactive map [2]
6. Print shapefiles
7. Print XML, Keyhole Markup Language or Geography Markup Language (GML) data [3]
8. Print GeoRSS data [4]

[1] Static Maps API V3 Developer Guide
[2] Google Maps Javascript API V3 Reference Website
[3] KML Documentation Introduction, https://developers.google.com/kml/documentation/
[4] GeoRSS.org

9. Print GPX data
10. Print GeoJSON data
11. Print Geo Microformat, [5] h-card Microformat [6] or Schema.org Geo markup [7]
12. Print geo meta tags to a webpage
13. Geocode and reverse-geocode an address
14. Obtain distance and directions between two points
15. Map Clustering and geohashing
16. Calculate additional properties of geospatial data
17. Calculate relationships of two geospatial features
18. Import geospatial data from and export to external repositories

Creating, updating and deleting geospatial data locally

Saving geospatial features locally can be as simple as saving the latitude and longitude data to two database columns. This is, in fact, what most Wordpress plug-ins do. Most often, the geospatial data that Wordpress needs to handle is the coordinates of where the post was written. A common Wordpress plugin will save one latitude, longitude pair and will save these numbers in the mysql database, in the wp_options table that contains a key/value pair of data for each post. The row will consist of the Post ID, the data title, for example one for "latitude" and one for "longitude", and the floating point number value.

Both Drupal and Wordpress use a relational database such as MySQL to store their object data in a structured manner. Drupal uses a far more intricate system called the FieldAPI and defines a special CMS field type (not to be confused with the mysql geospatial field type), called GeoField. Both systems define a method to store that data in the MySQL database, retrieve it, alter it and delete it. Drupal's GeoField module will be discussed more in depth in Chapter 4.

In figure 3.1 the data has been saved to the object using the GeoField module's input style of Well Known Binary, also covered in more detail in Chapter 4. The data was retrieved through an open source online repository http://thematicmapping.org/downloads/world_borders.php where the shapefiles were exported and the Well Known Text data extracted manually. The resulting POLYGON Well Known Text for the province of Vienna is not satisfactory as it is too general and leaves out large areas of Vienna, especially in the southern region. Creating the POLYGON manually using the Google Maps API or OpenLayers would create a more accurate map. The accuracy of Lower Austria is, on the other hand, much better.

Alternatively, data can be stored on the file system as well as in a database. Often times, the data is saved in XML, KML or GML format on the hard disk where the file name is either saved

[5] Geo Microformat website
[6] h-card Microformat website
[7] Schema.org Geo property website

to the database or it is referenced using a custom logic (such as using the ID number of the data object for the file name in a specific folder). Creating, updating and deleting individual files that are independent is much less complex than managing references between files, but it can reduce flexibility and decreases in efficiency as more complex lookups for the data need to be done.

3.2 Display the raw geospatial data on a web page

Use-cases that require the raw data to be printed to the web page often show data in tabular format. This is exemplified when listing points on a line and their coordinates as showin in figure 3.2. Other than the decimal, often the degrees-minutes-seconds data is printed to the page or a different data display method, but the data is most often saved as a floating point value in the database and converted during display.

In order to switch between data formats, Web-based CMS Systems use a variety of methods. One of the simplest solutions is to save the various formats, decimal, degrees-minutes-seconds or radians to the database so that conversion need only be done during the creation and update process of the object. Other solutions are to save the data in a single format, most often decimal, and then convert the data as needed during page view. Many additional caching layers, such as Drupal's separation of object load and object view phases are further able to speed up data rendering by saving the view data temporarily for a certain length of time.

3.3 Display a geospatial point on a static map

Static maps are a very light weight method of creating non-interactive maps using a third-party service. The maps are simple image files, usually in JPEG or PNG format, and can be saved on the local file system or in the database. Due to the fact that static maps do not change except when the parent object changes, the file must only be generated once on each change and the process does not require an external javascript library. Therefore, the method is very efficient.

Google Maps offers an API[8] to obtain a static map PNG file by referencing a url. That url can be called directly from the object using an <img> tag source that references the Google Static Maps URL, or by saving the image file locally on the server. Figure 3.3 shows the result when calling the PHP curl function on the following URL using the Drupal Geolocation module.

```
http://maps.google.com/maps/api/staticmap?sensor=false
&zoom=7&size=300x300&format=png8 &maptype=roadmap&
markers=size%3Amid%7Ccolor%3Ared%7C48.19884338049882%2C16.37034773826599
```

Other than the setting "sensor = false", which is required for our use case, the other query parameters can be chosen dynamically and as needed. The Google API's tend to provide many customization settings and options. For example, the color of the marker can be changed dynamically using RGB values passed in as parameters of the url, the output file format can be jpeg or gif, and the maptype can be set to satellite or roadmap or terrain.

[8] Google Static Maps Documentation

The resulting PNG file from the previous request for a 300x300 pixel image is only 48KB in size, which is very efficient.

Display HTML5 geolocation

Alternatively to static mapping, there are more dynamic services such as the HTML5 geolocation specification, which is a method of returning the device's current location using methods such as IP location lookup or even GPS. Web-based CMS technologies such as Wordpress and Drupal usually use the IP-based form of lookup.

Most modern browsers support the HTML5 geolocation specification [9] and they implement it by calling the following function:

```
if (navigator.geolocation)
{
 navigator.geolocation.getCurrentPosition(showPosition);
}
```

Several restrictions have been placed on the specification so as to ensure security and to make certain the user of the device is aware that he or she is sending the position to a 3rd party. The specification requires that the user is both alerted to the fact that they are being requested for the information and that the browser must wait for the user to accept before moving on in the browser rendering. This behaviour ensures that web services are unable to circumvent the request and determine the coordinates anyway.

The call to navigator.geolocation returns the current coordinates of the device and can therefore be plugged in to other technologies such as a dynamic Google Map or saved to the database for use in other applications and on other pages. [Hol11]

3.4 Display a geospatial feature on an interactive map

Much more common than static maps is the use of dynamic maps; maps that can accept input and interact with the user. Examples of these technologies are Google Maps API v3, OpenLayers and Leaflet. The libraries are loaded by embedding JavaScript files in the page.

Most interactive maps are based on the concept of slippy mapping, also called slippy maps, which uses a rectangular viewing area called a viewport. The viewport is filled with small image files, usually 256x256 pixels, called *tiles*, that display the physical landscapes such as mountains, rivers, and land masses.

Tiles are served using ajax requests made to an external server or an external service. A server dedicated to serving tiles is also called a tile server. Google Maps provides 4 base types of tiles: hybrid, roadmap, satellite and terrain. There are services that allow for custom tiles to also be created.

[9] HTML5 Geolocation Specification

Features are then provided using a vector server. Objects such as streets and boundaries are rendered on top of the viewport using vector and raster data. [Dab13] [Zha07] In the case of the Google Maps API roadmap, routes are added on top of the tiles using additional vector data provided by the service. Features data is provided by the feature server and then the client machine must process these features and render them on top of the tiles using javascript. Because the website stores the geospatial data that it wants to display on the server, that server is called the feature server. Of course, the server could use an external service as well to obtain features.

Additional JavaScript actions can be defined when the user triggers a javascript event such as a click, touch or zoom, by interacting with the viewport. For example, panning the map left or right will request additional tiles from the service for the new regions that are within the viewport. Moving the viewport can also mean new features will be retrieved from the feature server according to the bounding box of the viewport's updated viewing area. [10]

The viewport is also known as a bounding box, which can be defined as the box created by two (or four) latitude/longitude points. An alternative method of defining a bounding box is using the lines of latitude and longitude for the top, bottom, left and right sides of the box.

3.5 Print shapefiles

Rather than displaying data, there are many different methods of saving the data on the file system in text format. The Esri Shapefile specification is an output format used for geospatial data exchange. It is a vector data format and can be used to describe any geospatial feature in a Cartesian coordinate format. The output format is widely used because it is a relatively compact form of data, can be saved to the file system in .zip file bundles, making them highly portable, and the software that supports shapefiles, such as the popular ArcView GIS software for which the specification was first created has shown that shapefiles work well to both display and share geospatial features.

Geospatial objects are stored as a collection of shapefile files. Three files are mandatory: .shp, which saves the feature geometry itself, .shx, the shape index format so that the feature can be indexed for use in searches and navigation by client programs, and .dbf, a file to specify additional attributes of the geometry. .dbf files are saved in dBase IV format and are therefore able to be opened by spreadsheet software. Non-spatial data is saved in the .dbf file. A common example is the location's store hours or a description of the type of object being saved, such as a store, a car, or a meeting point.

Additional files, such as .prj, can be added to specify the projection that the data is saved in. Other file extensions are used for special indexing and metadata. These files are .prj, .sbn and .sbx, .fbn and .fbx, .ain and .aih, .ixs, .mxs, .atx, .columnname.atx, .shp.xml, and .cpg files. [-98]

Individual geospatial features described within the shapefiles must be saved in the same order across all files, otherwise the data is not valid. The first geospatial feature in the .shx file must correspond to the first geospatial feature in the .shp and .dbf files. Maintaining a valid relationship between the separate files is very time consuming and any errors in the program can quickly lead to corrupt shapefiles.

[10] https://developers.google.com/maps/documentation/javascript/reference

Some mapping technologies such as OpenLayers, Google Maps and Google Earth are able to render objects from Shapefile input, making it a rather flexible method for managing geospatial data. Many private as well as government organisations, including data.gv.at, make their geospatial data available in shapefile format (for example as a download on their website) .

3.6 Print a Keyhole Markup Language (KML) or Geography Markup Language (GML) file

KML [11] and GML [12] are XML extensions used to express geospatial data. Both languages are fairly similar. GML was created by the Open Geospatial Consortium (OGC) and based widely on the Well Known Text data structure defined by the OGC. KML is extremely popular due in part because it is the standard that Google Earth and Google Maps use to exchange data, usually to input a KML file that is to be rendered in a Google Earth map. To facilitate transfer of data, .kml files are more often zipped and given the extension .kmz. GML files have either .gml or .xml.

Web-based CMS's often use KML/KMZ and GML files as attachments or inside of their data if they are being referenced. Fields do not often store data in KML/KMZ/GML format because it is less efficient than storing data directly in the database. Storing KML and GML is more effective when the data needs to be returned because a client program requests it.

The Vienna University of Technology would look like the following in KML:

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
  <Placemark>
    <name>Vienna University of Technology</name>
    <description>The location of the Vienna University of
    Technology</description> <Point>
      <coordinates>16.3703,48.1988,0</coordinates>
    </Point>
  </Placemark>
</kml>
```

Notice that in the example, the coordinates appear reversed as longitude and then latitude (with the optional 0 for the altitude); they are comma separated with no space between the values. The position is expressed as longitude then latitude to be consistent with the Cartesian coordinate system of mapping. KML represents coordinates in an (x,y,z) coordinate system, therefore the listing is longitude (east-west or x-axis), latitude (north-south or y-axis) and altitude (z-axis). Altitude is described as meters above sea level.

The equivalent example in GML:

[11] http://www.opengeospatial.org/standards/kml/
[12] http://www.opengeospatial.org/standards/gml/

```
<?xml version="1.0" encoding="UTF-8" ?>
<xmlns:gml="http://www.opengis.net/gml">
<items>
   <Item>
      <name>Vienna University of Technology</name>
      <description>The location of the Vienna University of
   Technology</description>
      <where>Vienna University of Technology</where>
      <position>
         <gml:Point srsDimension="2">
            <gml:pos>16.3703 48.1988</gml:pos>
         </gml:Point>
      </position>
   </Item>
</items>
```

KML was developed to assist visualisation of geographic features on map or a globe and includes elements for map interaction such as viewport, altitude and zoom which is not specific to the object being viewed.

Among other tools that can render KML files are the services Google Earth, Google Maps, Microsoft Virtual Earth, NASA World Wind and ArcGIS Explorer. These services have also been dubbed "geobrowsers". In a paper written by Sandvik [San08], geobrowsers are defined as services that are

> ...capable of accessing georeferenced data over the internet, and the view can be two and/or three dimensional. A geobrowser can be a desktop program or an application embedded in the web browser.

In 2008, Google Earth was the only geobrowser to support the full feature range of KML features. Google Earth had intermediate support and all other services evaluated in the paper offered a basic support of KML input. OpenLayers has since then expanded its support of KML input data. [San08]

Both GML and KML can express more advanced objects such as polygons, linestrings and (geo)collections.

Although the KML specification was formally proposed to the OGC in 2007 after being developed for Keyhole's Earth Viewer and later Google Earth, the specification was more popular because GML was not developed as a visualisation language and cannot be used to visualize the data on maps as well as can be done by KML. [San08]

3.7 Print GPX data

GPX is yet another file based output format for geospatial data that is also XML based but it is much lighter weight than GML and KML. Therefore GPX is a preferred data transfer method for devices that require less complex data structures than GML and KML. The GPX output type is also supported in Drupal's GeoField module. An example:

```
<gpx creator="geoPHP" version="1.0">
  <wpt lat="48.198843380499" lon="16.370347738266"></wpt>
</gpx>
```

Additional properties for elevation, time, description, etc. are available to display both spatial and non-spatial data within the <gpx> tag. Due to it's nature however, this form of output is not widely supported within Web-based CMS's.

3.8 Print GeoRSS data

RSS is a specification for XML files that is widely used for the exchange of data. RSS files are also referred to as RSS Feeds. They define a limited number of attributes that may be attached to an item and as such are light-weight files that can be used to exchange limited amounts of data. They are most widely used to share summaries of entries on a website with other services and websites, providing a link to the original content site. [Ude08]

Many CMSs support RSS Feeds out-of-the-box including Wordpress, Drupal and Zend. The current specification of RSS is 2.0 [13]. Very similar to the RSS specification is the ATOM Feed specification [14] which is also an extension of XML and can also be used in combination with the RSS specification in a single document. Wordpress, for example, prints feeds that are both valid RSS and ATOM feeds.

GeoRSS is the geospatial extension for RSS which adds custom tags to the existing RSS specification to define geospatial features in Well Known Text format (see chapter 4). To add GeoRSS support to an RSS document, the GeoRSS schema must be loaded using the xmlns command in the beginning of the document as follows:

```
<rss xmlns:georss="http://www.georss.org/georss" >
```

Individual elements, known as <item>s in the document are then allowed to possess geospatial features. For the Vienna University of Technology, the GeoRSS Point data looks like the following:

```
...
<item>
  <title>Vienna University of Technology</title>
  <link>...</link>
  <pubDate>...</pubDate>
  ...
  <geo:lat>48.198567</geo:lat>
  <geo:long>16.369651</geo:long>
```

[13] http://www.rssboard.org/

[14] http://www.atomenabled.org/

```
  <georss:point>48.198567 16.369651</georss:point>
  <georss:featurename>Wien, Oesterreich</georss:featurename>
</item>
```

GeoRSS has two specification types, simple and GML. The above example is a simple GeoRSS example. Simple GeoRSS is designed to be as concise as possible with only the georss:<feature>, above georss:point, tag being required. All other tags are optional. The most common Simple GeoRSS features are POINT, LINE and POLYGON.

The second type of GeoRSS is the full GML extension which currently supports the GML 3.1.1 specification [15]. The above example would look like the following for the GML GeoRSS specification:

```
<georss:where>
  <gml:Point>
    <gml:pos>48.198567 16.369651</gml:pos>
  </gml:Point>
</georss:where>
```

Web-based CMSs do not often use the GML GeoRSS specification because the simple specification covers the majority of the use-cases for exchanging data.

3.9 Print GeoJSON data

JavaScript Object Notation, or JSON, is another specification for data exchange. It has a very simple structural format to describe both objects, denoted by start and end curly brackets, and lists, denoted by square brackets. Key/value pairs are enclosed in double quotes and separated by a colon. The basic JSON specification has both numeric and string data types. For a full explanation of the specification, see http://json.org.

GeoJSON is a JSON extension used to describe geospatial features by adding the geometry object to JSON. Currently, the GeoJSON supports the follwoing geospatial features from the OpenGIS Geometry Model: Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon. GeometryCollections represent lists of geometries.

The example location of the Vienna University of Technology would be:

```
{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [16.3697, 48.1986]
  },
```

[15] http://www.georss.org/gml.html

```
  "properties": {
    "name": "Vienna University of Technology"
  }
}
```

3.10 Print Geo Microformat, hCard Microformat or Schema.org markup

> Designed for humans first and machines second, microformats are a set of simple, open data formats built upon existing and widely adopted standards.
> - Microformats.org [16]

Microformats, as well as Schema.org schemas, simplify the process of structuring data in an HTML document. They present a standardized framework embedded in HTML4 and HMTL5 document tags, is a set of classes and tag attributes that lend context and readability to otherwise ambiguous data.

The motivation behind microformats is to format data such that humans are able to read it and understand it first. A secondary aspect is that the microformat gives the data, and a machine reading it, a context.

> ... while the objective of microformats is to enhance user experience, microformats are first detected by XML parsers, and provide explicit, non-ambiguous, machine-interpretable semantic information about the content they are attached to. [Mri08]

All microformats have the same rules so that the structures remain unified, valid XHTML markup and remain easily understandable to humans. The rules state that existing schemas are to be re-used as much as possible and to use well supported standards. The Microformats organisation has outlined the exact rule specification on the Microformats.org website.

Certain Microformats and Schema.org schemas are gaining widespread support, where services such as Skype and Google will identify the mark-up automatically and handle them in a context appropriate method. Skype does this by highlighting telephone numbers and creating a link on top of the number so it is dialable using the skype program. Currently the Google search engine will recognize images in the Schema.org (cooking) recipe schema and display the first image in result listings in its searches. If the geocoordinate schema becomes widely adopted, then Google may implement rendering the schema as a map image in its result set.

The Geo Microformat is responsible for geospatial data and is a direct result of the vCard and nCard specifications, which were developed for other software. vCard was already a widely accepted format for data structuring that included geospatial data. Microsoft Outlook, for example, uses the vCard specification for address book contacts.

[16] Microformats.org website

The most basic form of the Geo Microformat is the following:

```
<div class="geo" title="machine readable title">
  some human readable title
  <abbr class="latitude" title="decimal representation">readable
     latitude
  representation</abbr>
  <abbr class="longitude" title="decimal representation">readable
     longitude
  representation</abbr>
</div>
```

Here is a working example:

```
<div class="geo" title="Vienna University of Technology">
  Vienna University of Technology
  <abbr class="latitude" title="48.1988">48 11' N</abbr>
  <abbr class="longitude" title="16.3703">16 22' E</abbr>
</div>
```

The example would be printed directly on the web page. A typical display style for the above is: Vienna University of Technology: 48 11' degrees N, 16 22' degrees E.

Schema.org's geo schema specification is very similar and the above microformat example would be the following in Schema.org's geocoordinate schema:

```
<div itemscope itemtype="http://schema.org/Place">
Vienna University of Technology
<div itemprop="geo" itemscope
    itemtype="http://schema.org/GeoCoordinates">
    Latitude: 48 deg 11 min N
    Longitude: 16 deg 22 min E
    <meta itemprop="latitude" content="48.1988" />
    <meta itemprop="longitude" content="16.3703" />
</div>
```

Elevation is also available and its content is also a decimal number, but Schema.org does not specify any basis for the number. Elevation 0.0 is different from country to country, so the elevation of several GeoCoordinate objects may not be exact relative to each other. Schema.org allows additional non-spatial data to be attached to properties of the GeoCoordinates object such as name, url and image.

3.11 Print geo meta tags to a webpage

Meta tags are used in HTML documents and pages to add information about the document that is explicitly used by the browser or the device. It is normally not viewed by the readers of the page. Meta tags are embedded inside the <head> tag of a document and always possess a name and content attribute. The following is an example including geo tags:

```
<meta name="title" content="Vienna University of Technology" />
<meta name="description" content="A set of generated maps for the
    Vienna
University of Technology" />
<meta name="keywords" content="maps,university,vienna,technology" />
<meta name="geo.region" content="AT-9" />
<meta name="geo.placename" content="Wien" />
<meta name="geo.position" content="48.198655;16.368463" />
<meta name="ICBM" content="48.198655, 16.368463" />
```

Non-geospatial tags include title, keywords, and canonical (the canonical url). Geo Meta Tags include four properties, all of which are optional:

1. Region: This is a value taken directly from the ISO-3166-1 specification for countries and regions. In the previous example, AT stands for Austria and 9 stands for the 9th province (sorted alphabetically) which is Vienna.

2. Placename: Usually generated through analysing the text or auto-generated by lookup using an API such as http://www.getty.edu/research/tools/vocabularies/tgn/

3. Position: The latitude and longitude in decimal format separated by a semi-colon.

4. ICBM: A different form of the position element where latitude and longitude are separated by a comma and a space. ICBM stands for intercontinental ballistic missile and is used by some devices in place of the position tag.

Wordpress will auto-generate the non-geospatial meta tags, but plugins are available to include geospatial meta tags. Several plugins deal with adding geo meta tags directly to each post such as Geo Tag [17].

3.12 Geocode and reverse-geocode an address

Geocoding is the process of converting a human-readable address string to geospatial data that can be expressed in numeric form, usually as a latitude/longitude pair. Take the typical use-case where a CMS user wishes to enter address data into the website (for instance in a textfield on a

[17] Geo Tag: http://wordpress.org/plugins/geo-tag/

form) and this data is then sent to an external service where the text is turned into a latitude/longitude data pair. Geocoded data can be displayed on the page, or more commonly, the CMS is set up to store that data locally so that the service does not have to be contacted again until the data changes.

In the following example the CMS user has entered the address for Vienna University of Technology, Karlsplatz, Vienna, Austria. The string "Karlsplatz, Vienna, Austria" is sent to the geocoder API by means of JavaScript and the Google geocoder service returns a JavaScript array with (48.1988, 16.3703).

```
var geocoder = new google.maps.Geocoder();
geocoder.geocode({'address': 'Karlsplatz, Vienna, Austria'},
    function(results, status){
    //the data will be in results[0].geometry.location
    var point = results[0].geometry.location;
    alert('The service returned latitude/longitude of ' + point.lat()
        + '/' +
    point.lon() + '.');
});
```

Google's geocoder object is part of the Google Maps API and therefore loaded by embedding the Google Maps API JavaScript in the website. A website can be programmed to respond to the results obtained, for example, by saving the data to the database or by creating a slippy map with the returned data.

The Google geocoder service will return the determined latitude/longitude pair in result.geometry.location. The geospatial feature that was found is in result.geometry.location_type, which is of data type GeocoderLocationType. Several attributes of the GeocoderLocationType object will specify if the precise location was able to be found, or if only an approximate location was found.

Google's geocoder service goes beyond simple geocoding, which can also be used as additional information. For example, it will return the recommended viewport and minimum bounds to display the object being represented, which can also be saved to the database and used to initialize the slippy map's viewport.

Reverse geocoding is the process of turning a latitude/longitude pair into the address information. Geospatial data, such as (48.1988, 16.3703) is sent to the reverse geocoder service and the service returns an address string such as "Vienna University of Technology, Karlsplatz, Vienna, Austria"

In web-based CMSs such as Drupal, the geocoding is a highly requested use-case that involves the interaction of several fields. For example, an address field will save the human-readable address data and automatically retrieve the geocoded data from a geocoder service, prepopulating that data into a geospatial field (GeoField). [18]. The data must be coordinated between the multiple fields when creating, updating or deleting the entity.

[18] Drupal.org Geocoder Project Page and Geocoder documentation

3.13 Obtain distance and directions between two points

Obtaining driving directions between two locations is a difficult task facing web-based CMSs. GPS devices support driving direction and real-time traffic as part of their service. Web-based CMSs need to interact with external systems to obtain driving directions and traffic information.

Google provides an API [19] that can be used by a CMS to obtain driving directions. It provides two methods of output, JSON and XML, which can then be plugged into a mapping technology or passed on to other services. Google Direction API requests are made using HTTP requests to the Google service, not JavaScript requests. See figure 3.6 for an example of driving directions rendered on a Google map.

Driving directions provide complex challenges for CMSs because directions are based on travelling along several edges that connect end points. A single driving direction is called a route. Services are often interested in calculating a route's distance and tavel time so as to maximize efficiency, reduce travel time and to save on energy [Yua10]. These interactions can often be complicated and each website may have its own use-case for the data.

3.14 Map clustering and geohashing

Clustering is the task of grouping unlabeled data in an automated way [Dab13]. Geoclustering is the process of grouping geographical data in an automated way. Most often, this includes grouping geographical features together into one feature that share a similar trait, such as one marker on a map that displays the number of locations, or geospatial object, that lie within a certain region, or within a certain distance of the marker.

Geoclustering makes maps more reader friendly. Objects that are physically close to each other can often overlap and become "unreadable" if there are too many markers around it. When viewing locale types in a region, often these locations are physically too close to each other to be distinguished at a very high zoom level. See figure 3.7.

The GeoField module in Drupal makes use of a special algorithm, known as the K-Means algorithm, and the Euclidean distance 2 calculation method to cluster objects based on the current zoom level and viewport. Clustering is therefore performed on-the-fly and is fairly resource intensive.

To make the algorithm as efficient as possible, the Geocluster module uses a Geohash based cluster approach. A Geohash is a string value that is computed from the latitude and longitude. The world is broken up into grid sections and each section is given letters. Each section is again broken down into grid sections and these are again given characters. The processes can be repeated as many times as needed.

Each character in the string represents the value for that level, thus, for The Vienna University of Technology, the Geohash value u2edhw2d8g8h01048080 means that the Geohash level 1 value is "u" and the level 2 value is "u2", level 3 is "u2e" and so on. Each level represents a smaller, or more precise, area. If the map is zoomed all the way out, items that are in the same level 1 will be grouped together. Zooming in causes the items to be re-grouped by level 2, then level 3, etc.

[19] The Google Directions API Documentation Website

The Drupal Geocluster module, which makes use of saving Geohashes in the MySQL database, has been shown to effectively cluster up to 100,000 point data in 600ms [Dab13] on a typical server, which is tolerable. It is a rule of thumb today that websites should have their data available for the http request within 200ms of the request start, meaning the algorithm is still slow for modern websites. Services dedicated to searching, such as Apache Solr, have been shown to return clustered datasets much faster.

3.15 Calculate additional properties of geospatial data

Additional use-cases require knowing more about the geospatial data that is saved, for example to know the geometry type of the data, whether it is a point (representing a single location), a linestring (which can represent a river, road or similar geospatial feature) or a polygon (representing an area, country, lake, etc.).

For locations (point data), methods that evaluate the latitude and longitude or return one of these values are used to determine if a given viewport should display the data.

Specific functions exist for linestrings as well. Length functions exist to evaluate the length of a linestring. A common use-case is to obtain the distance of a river or a road. Driving directions also break down the paths traveled into small linestring segments that are evaluated for distance separately. Individual points from a linestring and a polygon can be returned to determine start and end points, or if a linestring is closed or not.

Polygons, or data that concern regions, are evaluated for their area similar to how linestrings are evaluated for their length.

The objects Point, Linestring and Polygon will be covered more in depth in the database layer and OCG Geometry Model sections.

3.16 Calculate relationships of two geospatial features

Common examples of relationships are “return all locations that are in this city, state or country”. Is the Vienna University of Technology in the first district of Vienna (no), is it in the city/province of Vienna (yes) or is it even in the country of Austria (yes)?

Use-cases dealing with rivers (linestrings) and lakes (polygons) deal with questions like: “Does this river cross impact lake by either starting, ending or crossing it?” or “Does this country fully or partially encompass this lake?”

The Open Geospatial Consortium has addressed these issues by creating guidelines for functions that return relationship data. ST_Contains evaluates two geospatial features and returns true if the first feature completely encompasses the second feature. There is the equivalent for the reverse order ST_Within, as well as for determing the negation. MBRContains is another function that determines the Minimum Bounding Region of the first feature and evaluates whether or not the second feature is contained in the Minimum Bounding Region. MBRContains is often used for determining if the data should be returned for a given viewport. These functions will be further discussed in 4 .

3.17 Import geospatial data from and export to external repositories

The aforementioned use-cases for printing data are concerned with the sharing of data. Web-based CMSs are concerned with providing services so that other systems can read that data. In recent years, Web-based CMSs have been moving away from simple web-page based display of data and towards providing their information as APIs or as data repositories.

Earlier in the chapter, geocoding using Google's or Yahoo's geocoder service was discussed. This is just one of many initiatives that provide geospatial data. Many services, like Google, provide a basic package for free, usually up to a certain number of requests to the service per day. Once this limit is hit, the website is required to pay for additional use of the service.

The data.gv.at - offene Daten Österreichs is a publicly available repository of geospatial data that is part of a larger open data initiative. The goals of the initiative are: To create transparency, presenting citizens the ability to view what the government is actively working on; to participate, drawing upon the knowledge and input from the community to assist in improving the data, service efficiency and quality; to collaborate, assisting in the exchange of data by providing a platform for the free exchange of data. A similar initiative, which the Open Data Austria works with, is the Linked Data initiative. It is "a set of best practices for publishing and connecting structured data on the Web." [Biz09] Central to the Linked Data initiative is that data must be openly accessible, which is usually done through a URI to a data file that is hosted on an external website. Data is also strictly separated from the presentation. The Linked Data initiative is a project that was launched in 2007 to identify existing open license data on the web and converting these to RDF standards according to the Linked Data standards. These collections are then made public by linking to them on the website.

Data sets for the Open Data initiatives, including the Open Government Data initiatives and the offene Daten Österreichs, are saved and published in various formats inclding CSV, JSON, RDFa, XML (KML and GML) und MediaWiki XML format. Data sets are compiled by various companies and government bodies and presented in an open source method. Example data sets include taxi waiting points, public WC locations and political boundaries.

The data may be used by third parties as long as the software using the data is also open source and the data does not violate privacy policies. Adhering to privacy policies is the responsibility of the creator of the data set. Additionally, the source of the data must somehow be labeled on the site using the data. It is enough to state the source of the data and provide a link to the location of the original file.

An example of how the data can be imported into a Drupal 7 Website can be found in the tuturial: Importing KML file data into GeoField field instance. This tutorial only shows how the data can be imported at the time of the page creation or update. A more interesting use-case is to check the external files at regular intervals, e.g. once per day, and create or update data objects into the system.

There are a variety of tools that CMSs implement to read data from files. XPath parsers is one method to extract data from a file in a customized fashion and map it to local data objects. RSS aggregators are often native to some CMSs such as Drupal. These aggregators will extract all data from an RSS feed and import each item in the feed as a separate data object.

According to Rax et. al [Rax11] there is no lack of large geospatial data sets available. The efficiency of the external repository is often a pre-requisite for choosing the repository. Services that are reliable and fast are given preference over services that are not. Many websites are also willing to pay for premium services to increase performance and reliability. Geonames.org is one example of a geospatial repository that provides an API for data exchange as well as a premium service.

Drupal has several modules that plug into external services. One service is the Geonames geospatial database. Here the data is fetched using a custom module called Geonames to interact with the Geonames API. As part of the set-up, the website administrator must obtain an account and place that key in the website's administration section as well as in those fields that will be used to store and fetch data. Other modules synchronize various geocoder services with the CMS. These interactions are further outlined in Chapter 4.

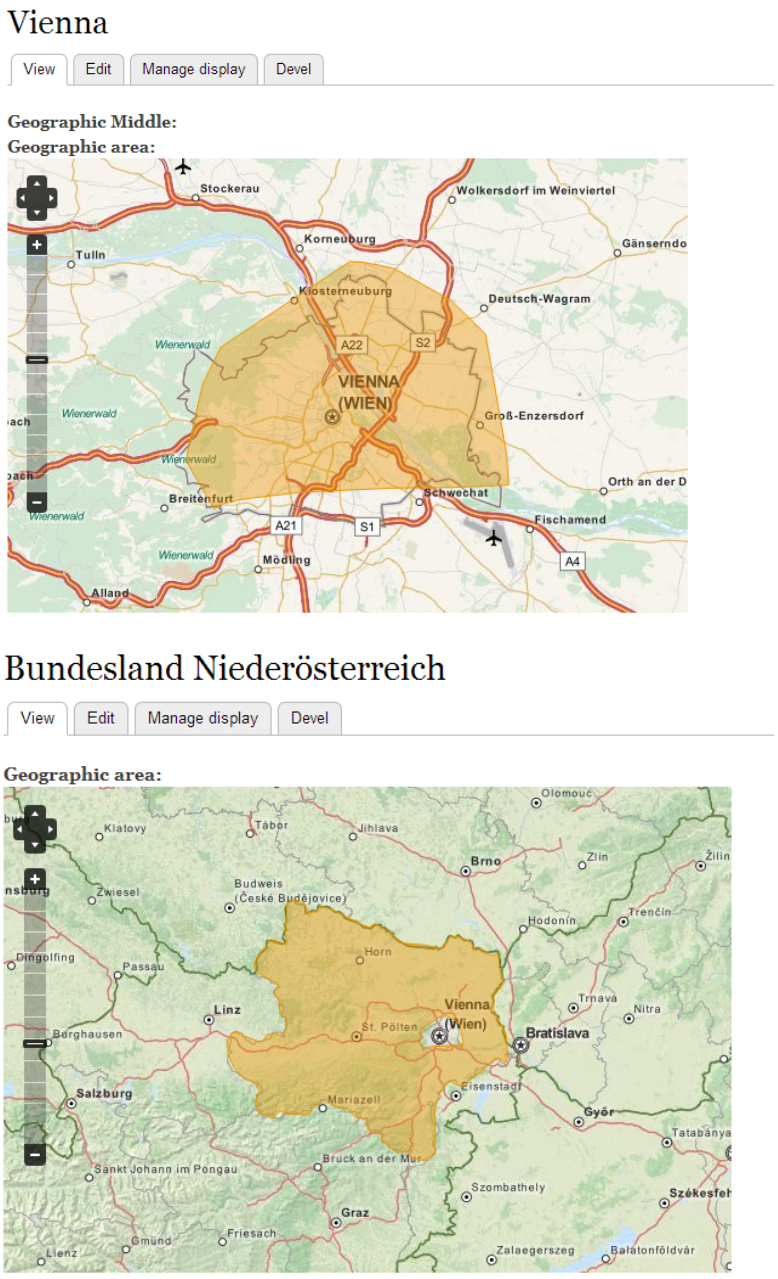

Figure 3.1: Displaying the input POLYGON, known as Well Known Text data, for the provinces of Vienna and Lower Austria on an OpenLayers map. The data is from an open source database, illustrating accuracy of data obtained from third party sources. The polygon for Lower Austria is much more accurate than for Vienna.

List of event locations

Title	Geographic Middle
1. Grazer Zaubertheater	Latitude: 47.095200000000 Longitude: 15.414300000000
1. Kärntner Erlebnispark	Latitude: 46.631200000000 Longitude: 13.437700000000
1. Tiroler Holzmuseum und Holzschnitzerei	Latitude: 47.401000000000 Longitude: 12.033100000000
1. Vorarlberger Bienenmuseum	Latitude: 47.191000000000 Longitude: 9.684400000000
1. Wander- Bauerngolf	Latitude: 48.051500000000 Longitude: 12.910800000000
1. Wiener Zaubertheater	Latitude: 48.217500000000 Longitude: 16.396000000000

Figure 3.2: The raw data printed to a page using a Drupal View and the GeoField module.

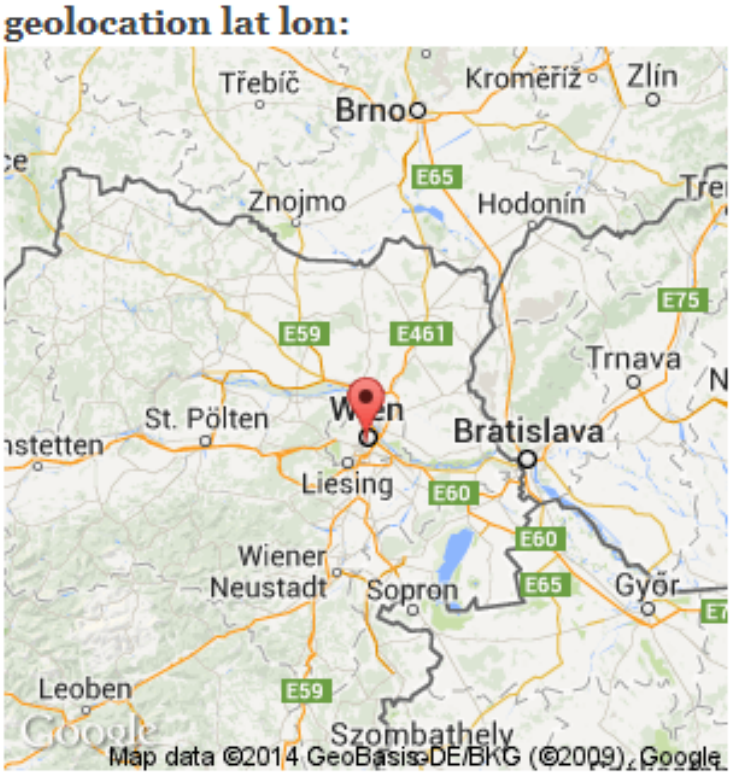

Figure 3.3: A generated static map jpeg file using the Google Static Maps API and the default settings in Drupal's Geolocation module.

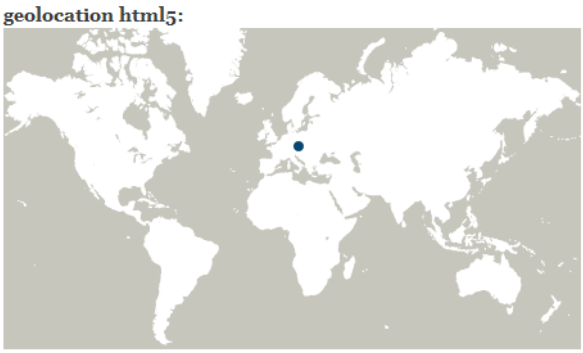

Figure 3.4: The default behavior in Chrome using the HTML5 Gelocation specification. A map is generated of the world and a blue dot signifies the position returned from the navigator object.

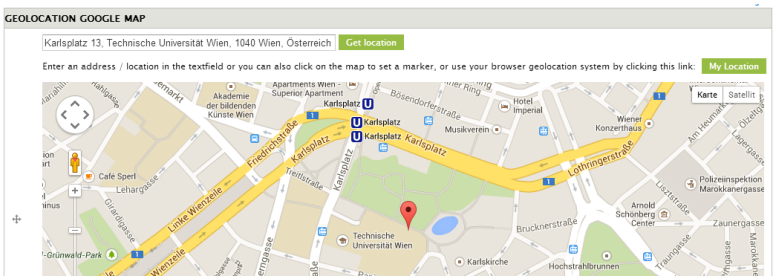

Figure 3.5: Using the Geolocation module in Drupal for geocoding a string. The input widget is a simple text field. When the button "Get Location" is pressed, an AJAX request is sent to the Google Geocoder Service. The method is more straight foward than the GeoField geocoder solution but is also less flexible.

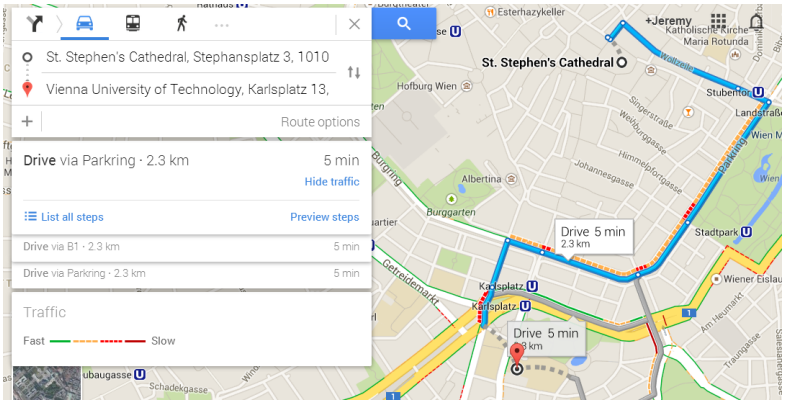

Figure 3.6: Driving directions provided by the Google driving directions service for travelling by car from St. Stephen's Cathedral to the Vienna University of Technology.

1 2 3 4 5 6 7 8 9 ... next › last »

Figure 3.7: Without using a clustering method, the data on this map looks cluttered.

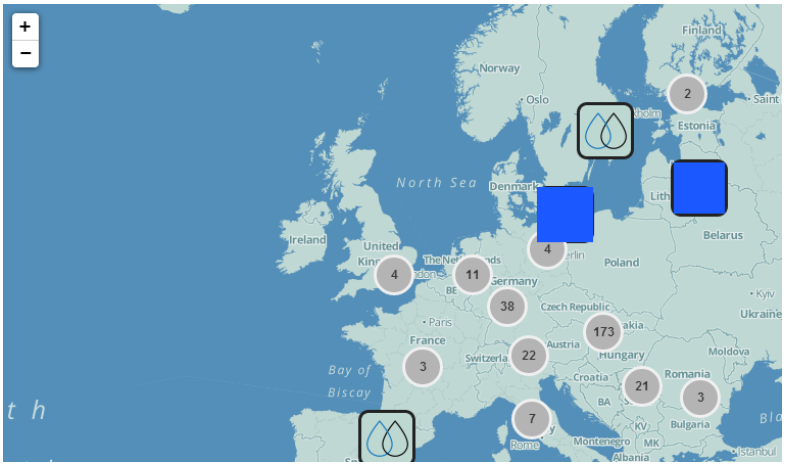

Figure 3.8: A map of the Drupalcamp Vienna attendees using the Geocluster and GeoField modules in Drupal

CHAPTER 4

State of the Art

Since the late 1990s geospatial specifications have been refined and standards for storing geospatial objects have been defined. Some of these standards have been discussed in the Use-Case section, see Chapter 3, but these standards have only started to be implemented in web-based CMSs in recent years. This section will discuss the CMSs and the standards more in depth.

4.1 Wordpress

Wordpress is currently the most popular web-based CMS holding a market share of 62% [1]. It is open source software that focuses on the administration of simple blog postings and the workflow rules surrounding blogging. Wordpress, however, does little to handle more complex data structures than a blog post. It provides good categorization logic, archivation and listings, but the geospatial use-cases that Wordpress presents a solution for are those that directly pertain to publishing blog content on a web-page.

Wordpress does auto-produce RSS version 2 feeds of its posts, but an additional plugin is required to print any geospatial data to the feeds. If the domain name of the Wordpress installation is http://www.example.com/ then the RSS is available at http://www.example.com/feed. The current RSS specification [2] 2.0.11 does not currently include a geo tag.

Recently an all-in-one solution for Wordpress, the WP Geo Plugin, [3] has been released. It covers a wide range of use-cases such as setting a post to a location, geocoding addresses, displaying (and customising) dynamic maps using Google Maps API v3. It will not, however, print out the data in any of the formats such as GeoJSON, RDFa, XML, GML and KML. There is an extensive list of geosatial plugins for Wordpress. See a more comprehensive list of Wordpress geospatial pligins on the listing page Wordpress Geo[4].

[1] As of June 2014. Source: CMS Market Share on Wappalyzer.com
[2] RSS Specification
[3] WP Geo Plugin Page
[4] https://wordpress.org/plugins/tags/geo

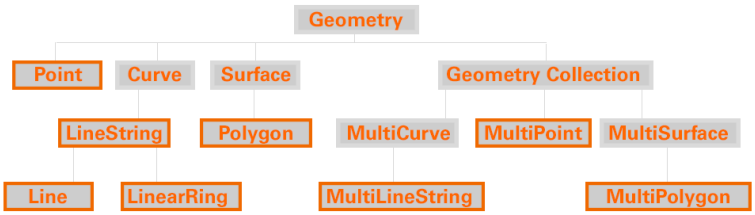

Figure 4.1: OGC Geometry Model data types

To deal with custom variables for blog posts, Wordpress adds a generic table to the database called wp_options, which saves a blog ID and key-value pair. Latitude and longitude values are written to the table separately as two rows. The value is always a VARCHAR and therefore the generic solution is unable to handle advanced queries on the database level. To compare latitude and longitude values against user supplied input, either the database data must be converted to decimal on-the-fly, or the value of all records needs to be loaded into php and converted using PHP.

4.2 The database layer and the OpenGIS Geometry Model

Other CMSs are moving toward more advanced database management, attempting to take advantage of geospatial extensions for which are based on the Open Geospatial Consortium Geometry Model. Most database engines, including Oracle, PostgreSQL/PostGIS, MySQL, DB2, SQL Server and Ingres, have at least basic support for spatial columns.

Geospatial columns are columns in the database that store geographical data in the same way that numbers and strings are stored in the database. In the implementation specification for geospatial columns from the OGC part 6.1.2. [Her10] the database table must specify which member types from the OpenGIS Geometry Model can be stored in the column, the Spatial Reference System ID along with other meta data. Most databases provide a couple of interfaces in order to save data to the column which includes functions to convert the data from Well Known Text and Well Known Binary (and reversed).

The OGC standard is known as the OpenGIS Geometry Model. It is used to model 2-D geographic features, where a geographic feature is defined as any real-world object that has a location. Geographic entites such as cities, routes, waterways, borders and countries. are all geographic features. The term “Geometry” has also been used by the OGC standard to refer to any geographic feature. The MySQL specific implementation for the OpenGIS Geometry Model also uses the terms geospatial feature and feature interchangeably. Therefore, in this chapter, we will also use the terms interchangeably.

The current release version of the MySQL community database is MySQL 5.6. The soon to be released MySQL 5.7 has made significant advances in supporting the OpenGIS Geometry Model and, unless otherwise specified, MySQL 5.7 is to be assumed as the version that is used.

The OpenGIS Geometry Model standard defines the following geospatial objects to represent geospatial features (see figure 4.1):

A Geometry is used to define any geospatial object in a 3-Dimensional environment. All subclasses of the Geometry class are defined as either 0-, 1-, 2- or 3-D objects in the environment and are used to define different geospatial features. Each Geometry has properties, such as coordinates, and definitions, such as simple or complex Geometries, or whether or not it possesses a boundary.

Points, the most basic objects, are 0-D instantiable objects that are used to represent locations such as cities, buildings, bus-stops or meeting points. A point is defined by its X, typically also longitude, and Y, also latitude, coordinate values and has an empty boundary.

Curves are non-instantiable 1-D objects represented by a sequence of Points and is classified as simple if the curve does not pass through a single point twice. It is defined as closed if its start point is equal to its endpoint. The boundary of a closed Curve is empty and in a non-closed Curve the boundary is considered its two endpoints. A Curve that is simple and closed is a LinearRing, which is a subclass of Linestring, which is a subclass of the Curve. LineStrings are defined as Curves with linear interpolation between two or more points. LineStrings are used to represent rivers, boundary lines, streets and routes and are defined by segments consisting of two consecutive points. It is a Line if it consists of exactly two points.

Surfaces are 3-D non-instantiable geometries, and Polygons are 2-D instantiable objects that are a sub-class of Surfaces. Whereas a Surface can include many planes, the Polygon resides on only a single plane. The only instantiable subclass of the Surface is the Polygon object. A Surface has exactly one exterior boundary, which is LineStrings, and zero or more interior boundaries, also Linestrings. The Polygon object is a Surface with multiple sides defined as segments. The boundary of a simple Surface is the set of closed curves corresponding to its exterior and interior boundaries. Polygon boundaries are defined as the set of LinearRing objects that make up both the exterior and zero or more interior boundaries.

There are several assertions that Polygons must hold to be valid. No interior boundary may cross a second interior boundary or the exterior boundary, and may intersect at a point as a tangent along the LinearRing. It may not possess lines, spikes, or punctures. A Polygon has an interior that is a connected point set.

Geometry Collections are simply collections of multiple Geometry objects outlined above. Subclasses restrain the type of Geometries that can be included in the Geometry Collection, but the Geometry Collection class makes only one assertion: All objects in the Geometry Collection must use a common Spatial Reference System (i.e. one coordinate system, e.g. latitude and longitude). Subclasses restrict membership: MultiLineStrings are a collection of Linestrings.

Although it is not often done, latitude and longitude is only a commonly used system for X and Y. When modelling features that are regional, a Point can be defined as (0, 0) and all (X, Y) values saved can be relative to it. Such a system can potentially save disk space as the maximum and minimum X and Y values can be restricted to less than than +/- 180.0 and +/- 90.0.

Well Known Text

Well Known Text is a specification for defining OpenGIS Geometry Model objects as simple strings. No other semantic or contextual information is saved in the string, such as the line color

or the type of object being represented (e.g. a mountain, a range, a building or bridge, etc.). There are currently eighteen keywords for WKT:

1. POINT
2. MULTIPOINT
3. LINESTRING
4. MULTILINESTRING
5. POLYGON
6. MULTIPOLYGON
7. TRIANGLE
8. CIRCULARSTRING
9. CURVE
10. MULTICURVE
11. COMPOUNDCURVE
12. CURVEPOLYGON
13. SURFACE
14. MULTISURFACE
15. POLYHEDRALSURFACE
16. TIN
17. GEOMETRYCOLLECTION

There are some differences between the MySQL, OpenGIS Geometry Model and the Well Known Text specifications that are worth mentioning. MySQL does not include TRIANGLEs, CIRCULARSTRINGs, COMPOUNDCURVEs, CURVEPOLYGONs, POLYHEDRALSURFACEs and .

A is a vector based representation for a surface and is used widely to store topological data of areas such as mountains and seabeds. It best illustrates change in altitude and depth for a structure, but is very expensive for storage space. are not often required for web-based CMSs, as they provide more detail than which usually must be presented to a reader. There are, however, many situations where the fine granularity can help, espeically when modelling terrain.

In Well Known Text, the POINT accepts two decimal values, an X and Y value. CURVEs and SURFACEs are defined as groupings of POINTs. Depending upon the type of geographical object being represented, it can be open - the end points do not touch - or closed - the starting

and ending points are the same. It can also be simple or complex. MULTIPOINTs are strings of POINTS. As it can be assumed that each (LON LAT) pair is a POINT, the keyword POINT is removed. LINEs are presented as a string of points where the first POINT is the starting location, and each subsequent POINT is then connected by a straight line.

Well Known Text can be used as an input method for geospatial columns in a database by using the function GeomFromWKT(<WKT>). GeomFromWKT is the MySQL function, PostGIS uses ST_GeogFromText(<WKT>). This difference in function names is one of the difficulties that prevent web-based CMSs from readily adopting geospatial columns.

Well Known Binary

The alternative to Well Known Text is Well Known Binary, which is a binary representation of the data and is a good format for dumping data from one database to another because it is more compact.

To convert from Well Known Text to Well Known Binary, the values are converted as either 32-bit (4 byte) integers or as 64-bit (8 byte) double-precision floating-point numbers and then serialized using one of two methods: NDR (little endian, least significant bit first) or XDR (big endian, most significant bit first). Specific bits in the sequence indicate the GEOMETRY type being saved, such as POINT or LINESTRING or POLYGON, the length of each Geometry item, the type of serialization and the values. [14]

Alternatively, Well Known Text data can simply be stored in string/blob columns in the database. This is what the Drupal Geofield module does, does due to the fact that the database abstraction layer is currently not able to communicate with true geospatial columns. Custom programmed solutions then convert the data to type geospatial on-the-fly as needed so that geospatial queries and functions can be executed.

As mentioned in Chapter 1, geospatial databases have geospatial data type columns that can store the data in an optimized fashion. How geospatial columns are to be defined by the is outlined in the manual [Her10] written by the OGC on how to define geospatial data type columns. Figure 4.2 displays the geospatial data types that are supported by MySQL.

In the specification, the column should save longitude (x), latitude (y) and bounding box max and min of both x and y values as decimal numbers that are optimized for the specified data type as well as the Well Known Binary data. More complex than the simple POINT, which has exactly one (x y) data pair, other geospatial data objects will have an variable number of (x y) pairs and therefore the table column will function similar to a varchar column type. The individual field size is relative to the complexity of the geometry being saved.

To illustrate the differences within a table column, a single table with POLYGON data has been filled using the default Well Known Binary in a MySQL blob column data type. This is the standard that Drupal's GeoField module does as is explained later in this chapter. If we take the data set that is used later in Chapter 5 for the timing tests, the table that stores POINT data has 6291 rows in the table and a table size of 5.5MB of data (measured using the MySQL Workbench). This includes the data used to store the entity ID, the field status, etc. By making a copy of the table, manually changing the WKB Blob data type column to a geospatial POINT data type column and using a script to export the data from the original blob column and re-import it using the MySQL GeomFromWKB function, the table size remains at 5.5MB. The

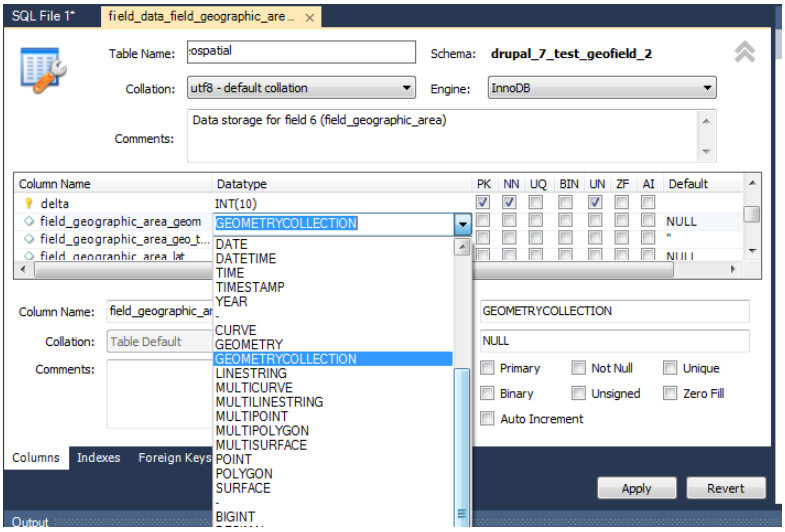

Figure 4.2: MySQL Geospatial Column Data Types

difference, however, is that the original columns for x and y, as well as top, bottom, left, and right can now be dropped because the geospatial column will handle that in an optimized way. Dropping these unneccessary columns shows in the MySQL Workbench table analysis that the data is now 2.9MB, a reducton of nearly 48%. The index length dropped from 4.0MB to 1.4MB, which is a 68.1% decrease. Chapter 5 will test to see if the geospatial column optimization can also improve the query efficiency of the table.

Geospatial Data Support Performance and Benchmarking

The performance of each database engine varies widely, and therefore tools have been created to benchmark database performance for geospatial data. Jackpine [Rax11] is one such program, written in Java, that is able to benchmark different databases including those mentioned earlier in terms of spatial performance.

The authors of Jackpine saw a need for an industry-wide spatial benchmarking tool for a range of databases due to the fact that the Transaction Processing Performance Council (TPC), an organization dedicated to creating benchmarks for various database use-cases, had yet to develop a satisfactory spatial database benchmarking tool. [Rax11]

Drupal also has custom benchmarking and debugging tools that are available using the Dev Module. It is mentioned here because Chapter 5 will propose methods to improve CMS interaction with geospatial database columns, but not with optimizing database performance directly.

4.3 The State of the Art of Drupal

The Drupal CMS has a set of methods for efficiently storing and flexibly displaying both geospatial and non-geospatial data as well as managing content workflow.

The current core version is Drupal 7.28 and, unless otherwise specified, that is the version being discussed. Drupal 8.0 is scheduled to release in Q4 of 2014, but for geospatial use-cases there is little support in the core distribution. For both Drupal 7 and 8, modules must be installed in order to cover the use-cases outline in Chapter 3.

Several Drupal specific terms are mentioned in this chapter. Nodes is the general term for an object that has at least a title and often a generic textarea called a "body". Additional data is saved in what is called "field instances". Nodes can also have a classification, called a Taxonomy, which groups similar content using keywords.

In order to discuss the state of the art of Drupal and its geospatial support, a few of the underlying concepts must also be outlined such as the FieldAPI, Render Arrays, Field input widgets and field display formatter.

The Field API and render array: data handlers vs. data formatters

> The FieldAPI allows custom data fields to be attached to Drupal entities and manages storing, loading, editing, and rendering of the field data. Any entity type (Node, user, etc.) can use the FieldAPI to make itself "fieldable" and thus allow fields to be attached to it.
> Api.Drupal.org about the Field API

As of Drupal 7, the FieldAPI is a part of the core distribution. An entity has field instances of field types, where field types are defined by Drupal modules using the FieldAPI. We will constantly go back to the FieldAPI when discussing modules and their function for fulfilling Geospatial use-cases. It is important to have a basic understanding of the FieldAPI.

Fields are defined by modules by invoking Hooks. Hooks are events that are invoked by the Drupal core system or by supporting modules. They evaluate all modules and themes for instances of these hooks and execute them in a pre-defined order. The FieldAPI provides a set of hooks so that a module can use them to declare new field types or interact with existing field types from other modules. A module declares a field by implementing at least hook_field_info, which is a function in the module that returns a strictly defined array that informs Drupal, by way of its FieldAPI, how to deal with the field type. Normally other hooks are required to properly structure the data before saving to the database or when displaying the data. For a complete list of defined hooks, see the Documentation on Hooks in Drupal 7.

The Drupal 7 Geolocation module, for example, uses the FieldAPI to define a field, called "geolocation_latlng". The module returns a single field description array defined in the following function:

```
/**
 * Implements hook_field_info().
 */
function geolocation_field_info() {
  return array(
    'geolocation_latlng' => array(
      'label' => t('Geolocation'),
      'description' => t('Geolocation input.'),
```

```
    'default_widget' => 'geolocation_latlng',
    'default_formatter' => 'geolocation_text',
   ),
  );
}
```

Drupal strictly divides data handlers, known as Field input widgets, and display styles, called field display formatters. The FieldAPI makes use of this by asking for the "deafult_widget" (data handler) and "default_formatter" (data display method) for each field defined in the module. Both the defining module and any supporting modules can provide additional field display formatters and Field input widgets.

The theory of a strict divide of content data storage and content data display methods was formally proposed by Reenskaug in 1973 [Ree73] where he outlined a framework for complex information systems to become module. It was the dawn of the MVC framework which would be formally proposed for Smalltalk-80 in 1999. The FieldAPI is one part of Drupal's instantiation of a MVC system.

Often the data saving method is dependent on the data display requirements. Suppose the goal is to display a Geo Microformat on the page, then it is inefficient to save the address information, such as street name, postal code and city name, and use a geocoding service for every page view to obtain the latitude and longitude coordinates. It would need a geocoder to first process the human readable address in order to obtain latitude and longitude coordinate points for the Microformat and save those to the database. Just this scenario, however, is what makes Drupal most flexible. Various input widgets allow for the data to be input in different formats, such as an address string and the geocoding done once. The default widget for the Geolocation module is a simple widget, which only accepts the decimal form of the latitude and longitude coordinates from two HTML textfield inputs and saves those values to the database.

Render Arrays were only marginally supported in Drupal 6 and were developed much more thoroughly in Drupal 7. When building a specific page, Drupal will go through all modules and core settings and ask for page data. If the page being viewed is a full text page for a Node that has a geolocation field instance, then the Node is loaded from the database where the FieldAPI tells Drupal to load the data for each defined field instance belonging to that object. An array of information is returned first. Here is the relevant part of the array in the above context for the full text page.

```
$render_array => Array
  [field_geolocation_field] => Array
    (
      [#theme] => field //<-- theme_field is responsible for this data
      ... other data ...
      [#title] => Geolocation Field
      [#access] => 1
      [#label_display] => hidden
      [#view_mode] => full //<--- our context
      [#field_name] => field_geolocation_field
      [#field_type] => geolocation_latlng
```

```
[#entity_type] => node
[#object] => stdClass Object
  (
    [title] => Vienna University of Technology, Karlsplatz,
       Vienna,
    Austria
    [field_geolocation_field] => Array //<-- the geolocation
       field type
      (
        [und] => Array
          (
            [0] => Array
              (
                [lat] => 48.199
                [lng] => 16.367
                [lat_sin] => 0.745466
                [lat_cos] => 0.666543
                [lng_rad] => 0.285663
              )
          )
      )
    [rdf_mapping] => Array
      (
        [rdftype] => Array
          ...
```

In render arrays, the # sign denotes elements that are key words belonging to the theme function that is currently being handled. They are items such as “#type”, “#title” and “#markup”, whereas elements without the # sign are either data elements relevant to the object being rendered (in the example above, this is “lat” and “lng”) or they are children (e.g. the “0” underneath “und” - which stands for “undefined language” - indicates that this is the first row of data for the geospatial field instance and is not of a specific language such as “de” or “en”).

Now that the object is loaded from the database, Drupal will cycle through all elements in the array for the presentation. This is known as the view phase. When it is time to render the “field_geolocation_field” field instance, Drupal navigates down the array and sends the child data to the “theme_field” function which determines the context. It then passes the field on to “theme_geomap_geolocations” because it was chosen as the display method for this context on the Drupal administration page. In this example, it will be displayed as a geo microformat (unless a module or theme that comes later in the chain changes this behavior). Finally, “theme_geomap_geolocations” in the Geomap module (which defines the Geo Microformat formatter for the Geolocation field type) will return the rendered information for viewing:

```
<div class="geo" title="Vienna University of Technology, Karlsplatz,
   Vienna,
Austria"> Vienna University of Technology, Karlsplatz, Vienna,
   Austria
  <span class="latitude" title="48.199"></span>
```

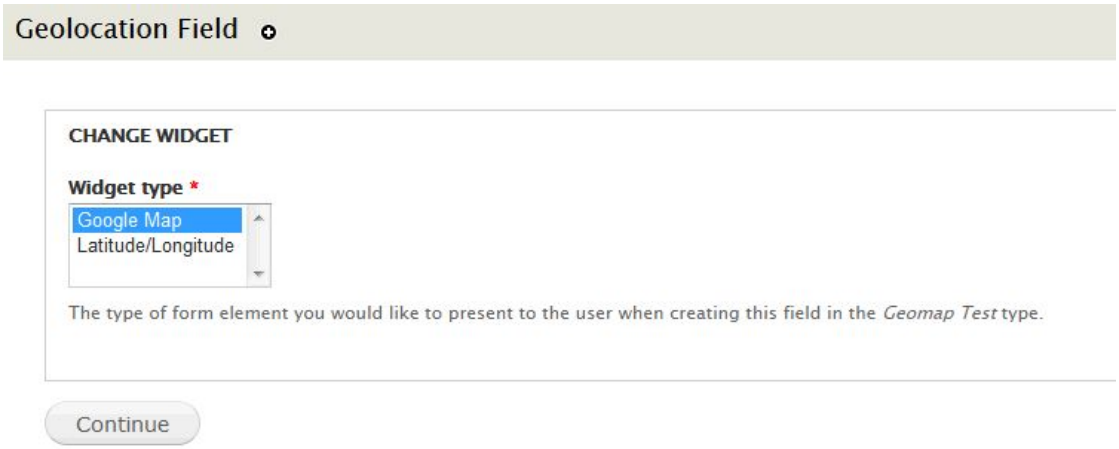

Figure 4.3: Widget types defined in the basic Geolocation module and the additional Geolocation Google Maps module.

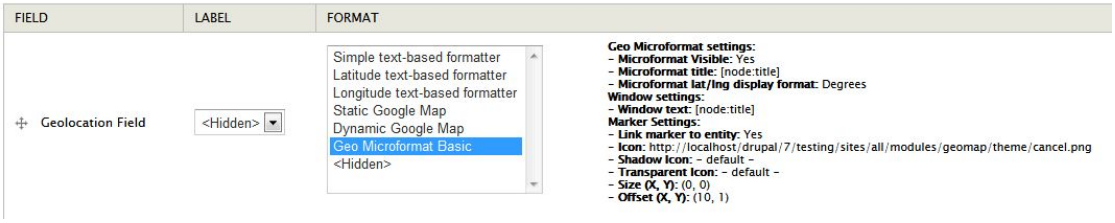

Figure 4.4: Handler types defined in the basic Geolocation module and the additional Geolocation Google Maps module.

```
  <span class="longitude" title="16.367"></span>
</div>
```

For detailed information as to how the FieldAPI and Render Arrays work, see one of the many online documentation pages or tutorials at API.Drupal.org

Geolocation module

The Geolocation `http://drupal.org/project/geolocation` module is a good example of a simple module that defines one field type, “geolocation_latlng”, with one basic field widget seen in 4.3, a pair of HTML textfields for latitude and longitude, as well as three separate display handlers , “geolocation_text”, “geolocation_latitude” and “geolocation_longitude”. A contributed module such as “geomap” then provides a special formatter for the data collected by the geolocation module’s “geolocation_latlng” field type.

The Geolocation module is primarily a data gathering module as shown above. The data is saved in the database locally as decimal values and it does not handle input in the form of

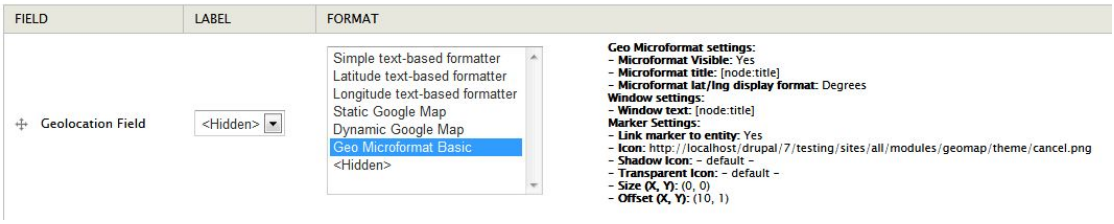

Figure 4.5: Handler types defined in the base Geolocation module and the Geolocation Google Maps module.

address information. However, the auxillary module Geolocation Google Map does provide an extra textfield for address strings that will be sent to the Google Maps API geocoder service. If the service is able to successfully retrieve the geocoded information, that information is saved to the database. The address string is discarded.

The display method provided by the Geolocation module uses the Google Static Maps API to display a point on a map graphic, which is saved as a JPG image file locally on the file system.

Geolocation also possesses a sub-module that utilizes the HTML5 geolocation gathering method to obtain the geolocation of the reader. The geolocation specification is then queried for the latitude and longitude value supplied by the reader's browser and this value is saved to the database. It can be used efficiently where the data is entered on-site.

Geomap module

The addition of the Geomap module allows for Geo Microformats to be disaplyed to the page and it also provides a JavaScript code snippet that will display all Geo Microformats in a Google Maps API v3 slippy map.

Geomap adds one FieldAPI field formatter for the "geolocation_latlng" field called Geo Microformat Basic with a machine name "geomap_fields_formatter_basic". The formatter uses theming, templating and the render array to print a Geo Microformats to the page. The module extends the basic Geo Microformats in that it uses additional markup embedded within the microformat to tell the Google Maps API what icon, window data and map settings the administrator desires. Due to how Microformats work, additional data is ignored by other applications not requiring it, as it is not part of the original specification.

Location module

The Location module is one of the oldest geospatial modules in Drupal. It was started in 2005, before the FieldAPI was developed, and was known at that time as the Content Construction Kit, a.k.a. the CCK, module. The Location module defines locations anonymously to the content. Content, known as both users and Nodes, can possess zero, one, or more locations, depending on how the administrator defines the location field settings. The advantage to this approach before the advent of the FieldAPI was that the Location could be saved once in a bundle and attached

to multiple Nodes. Since then, the approach has become outdated due to the arrival of newer modules, but many websites still report using the Location module.

A location for this module is defined as a latitude/longitude coordinate point with extra database fields for the human readable "location name": "Street address", "additional address", "city", "province", "postal code", "country". There is an aditional input for the source, indicating whether the location was entered manually or was the result of a service such as Google's Geocoder service.

To increase efficiency, the location module takes advantage of Drupal's native caching mechanisms in that it provides a cache table where it stores rendered locations. The FieldAPI has largely taken over the role of caching, however. Once a location is generated, the system no longer has to do the expensive work of re-generating a location the next time it is accessed. Rendered data can be found in the "cache_location" table.

Although the Location module has to adapt in order to use the FieldAPI, the Location module coveres such a variety of use-cases, meaning it is still meaningful for Drupal website administrators. It can display locations as simple text, or as locations on a slippy map or as GeoRSS feeds. The Location module also includes a geospatial function library written by Ka-Ping Yee. It uses the ellipsoidal model of earth and computes the shortest distance between two given locations (using the latitude and longitude coordinates). Computations are made using the PHP layer. The Location module also includes a custom search algorithm, which takes advantage of the Drupal Search API, instead of implementing a view plugin using the Drupal Views API, an approach that other modules, including GeoPHP, use.

There is no support in the Location module for specifying OpenGIS Geometry Model objects such as Polygons or Points.

Gmap module

The Gmap module is an auxilliary module primarily for the Location module, in that it renders a Google Maps API slippy map and provides several functionalities for other modules as a custom Drupal Google Maps API. It is a fairly robust module in that it contains several sub-modules to cover multiple geospatial use-cases. Its primary function however is to display the data gathered through other modules on the slippy map.

GeoField module

GeoField with the GeoPHP toolset is the most flexible solution for geospatial data management in Drupal which possesses the basis to make use of geospatial columns in the database. It is primarily a data handling module, but boasts several flexible display methods as well as the ability to search for geospatially related data, such as data in the same area and data within a certain geospatial distance on the PHP level, which is only available through the Location module otherwise.

GeoField defines a set of four input methods, the choice of which to use partially depends upon which geometry types from the OpenGIS Geometry Model are to be stored.

The four widgets that are defined by the GeoField module are:

1. Well Known Text
2. GeoJSON
3. Latitude and Longitude
4. Bounds

Both the Well Known Text and GeoJSON input methods provide a single HTML textarea input widget for data entry and can be used for any OpenGIS Geometry Model object. The latitude and longitude input method provides two textfield input boxes that accept only decimal values and will restrict the input to Point data. Bounds expects the decimal formats for top and bottom, latitude, and left and right, longitude, which will then be able to model Bounds.

In previous versions of the GeoField module, in addition to choosing one of the four input methods, the administrator is given the option to present all four input methods to the CMS user simultaneously, and allow the user to choose which option is best for that particular entity. The system then accepts the first input method that was filled out. It was an elegant solution for situations where the preferred input method would vary from entity to entity. This can be illustrated by the following example: The data source varies, sometimes being written in Well Known Text and the next source provides the data in decimal format. The disadvantage is that this method requires a higher level of knowledge on the part of the CMS user. The method is prone to errors as the input can often be invalid for certain OpenGIS Geometry Model data types.

Additional modules define other Field input widgets such as file uploader widgets that extract data from files. File uploading is a good method in which GeoField information arrives from various sources where an automated process is not feasible or where the processes should only happen irregularly. This is exemplified in a blog post where only a few items would require a GeoField value. The figure 4.6 shows how such a widget can be put to use.

The GeoPHP module has objects that represent each supported Geometry type, including the basic Point, Linestring and Polygon geometry types and the corresponding Multi-geometries. Although the GeoPHP library is a very elegant solution providing simple functions that are modelled after the OpenGIS Geometry Model, it is limited by the fact that many equations in PHP are difficult to program and they are currently not implemented. Determining relationships such as ST_Within or ST_Distance in not possible using the basic GeoPHP library. However, doing other functions are, such as determining the Minimum Bounding Region.

The GEOS php extension is a PHP library extension that models the OpenGIS Geometry Model data objects in binary PHP and has been shown to be ten times faster than the GeoPHP library. [5] Enabling the GEOS PHP extension is the most difficult step. There is no binary compilation package available for servers, so the library has to be compiled on each server, and there is currently no distribution package for Windows servers.

The GeoPHP Drupal module uses a flag function called geosInstalled, defined in the geoPHP.inc file, to determine if the GEOS library is installed. Without the extension, the module falls back to using GeoPHP. With GEOS, the system is able to mimic the database level functions that are outlined by the OGC geospatial standards.

[5] See Patrick Hayes' Wiki on GEOS

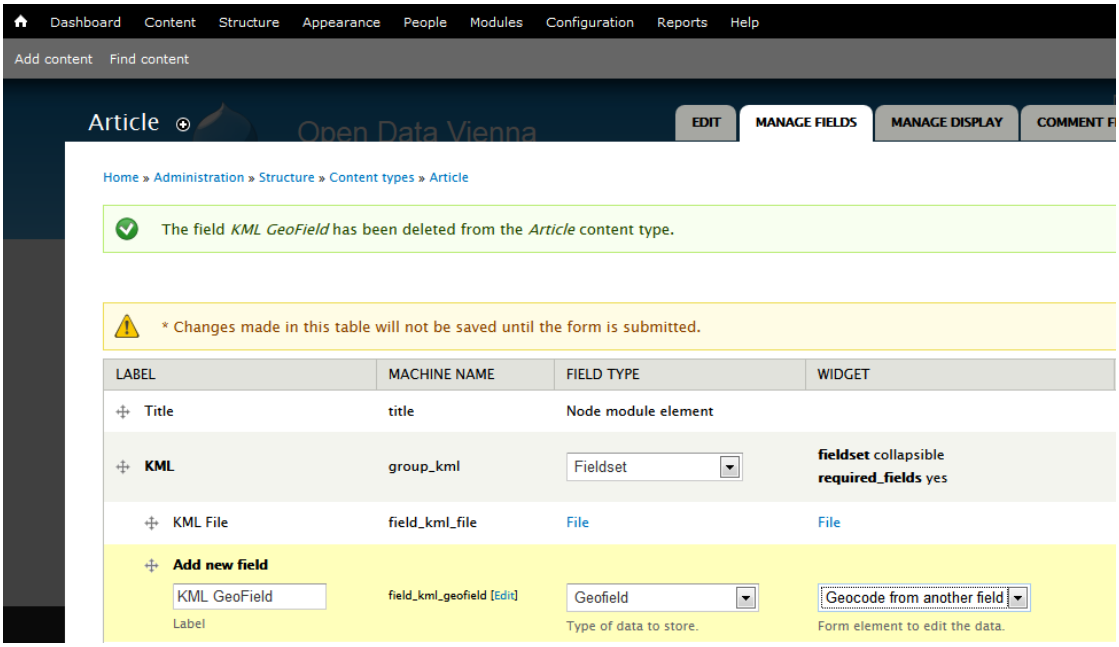

Figure 4.6: Creating a GeoField that will automatically import its data from a KML file field

```
public function length() {
  if ($this->geos()) {
    return $this->geos()->length();
  }
  $length = 0;
  foreach ($this->getPoints() as $delta => $point) {
    $previous_point = $this->geometryN($delta);
    if ($previous_point) {
      $length += sqrt(pow(($previous_point->getX() -
          $point->getX()), 2) + pow(($previous_point->getY()-
          $point->getY()), 2));
    }
  }
  return $length;
}
```

The GeoPHP library defines each OpenGIS Geometry Model object in its own class folder, but only a subset of the OpenGIS Geometry Model objects is made available:

1. Collection
2. Geometry
3. GeometryCollection

4. LineString
5. MultiLineString
6. MultiPoint
7. MultiPolygon
8. Point
9. Polygon

It will be shown in Chapter 6, what the performance difference is between using the GeoPHP library for some functions against using true geospatial columns in MySQL.

All GeoField widgets save the input data in a rigid format. Depending on the input widget and the type of data being saved (Point, Multipoint, Polygon, etc.), certain columns in the database will be the key data columns. The FieldAPI defines an entire table in the database in which to save the data stored in each GeoField field type field instance with columns for entity ID and other indicators for the field instance. The database columns responsible for the geospatial data for the field instance are:

1. Well Known Binary of type blob
2. Geo type - text
3. Latitude, Longitude - decimal
4. Left, Top, Right, Bottom - decimal
5. srid - integer
6. accuracy - integer
7. source - text

Geocode from another field

The Geocoder module in Drupal builds a bridge from the AddressField Module to the GeoField module. The AddressField field type field instance, which collects human readable address info on a per country basis, can be used as input that will be sent to either Google's, Yahoo's or a different Geocoder service. The geocoded result will automatically be written to the GeoField field instance. The fields and the choice of service are set per GeoField field instance and are therefore highly customizable. Normally in this case, the GeoField input widget is hidden from the CMS user and is automatically managed by the Geocoder module.

Geoclustering

The Geocluster module for Drupal applies the geohash mentioned in Chapter 3 to Point data in field instances of the GeoField module by adding one string column to the table for each geohash level. To do this, the Geocluster module takes advantage of several FieldAPI hooks that pertain to the create, update and delete events of both the field definition instance itself (e.g. creation of the field on the Node type) as well as on the individual field instance. Indices are placed on the geohash database columns so that queries on the data are efficient. The added columns make the query efficient, but the data storage is inefficient as each row entry requires ten varchar strings to be stored of increasing length.

Geocluster requires that lists be created so that it can fetch and group the data. Views API view instances must be configured properly to display Geoclustered data. To accomplish this, Geocluster has a dependency on the GeoJSON Feed which is a formatter for the Views API. Once a view is output as a GeoJSON Feed, the Geocluster module takes this as input and then turns it into map data on a slippy map. The script that updates the slippy map is a custom script that makes AJAX calls to the website on each event such as zooming in and out or updating the viewport.

CHAPTER 5

Methodology and Testing

5.1 Methodology of the Tests

Spatial columns in the database can shift workload away from the php level. As stated in Chapter 4, the GeoField module for Drupal does not take advantage of technologies at the database level because it cannot store the data in a geospatial column. The data is stored as Well Known Binary in a blob column.

Up until now, the state of the art has been discussed in detail for Drupal using the modules GeoField with GeoPHP to display sets of geospatial data and to filter them based upon locality or address.

There are theoretical reasons for making the change to geospatial data columns in the MySQL database, but a key factor is the performance gain. More precisely, the questions to answer are:

1. Does storing data in a geospatial column in the database opposed to a Well Known Binary Blob bring significant performance improvements?

2. Does storing the radian, cosine and sine values of latitude and longitude directly in the database bring a significant performance imrprovement?

A testing framework was set up to compare the geospatial functions ST_Contains, ST_Within, MBRContains, MBRWithin, Area and the Haversine Great Distance formulas. Timing tests were recorded to compare the efficiency of several queries between loading the entire object as a PHP Object and that of making the computations using the GeoPHP library where availably as opposed to querying a geospatial column in the database. Querying a geospatial column was done using two methods, converting the Well Known Binary data column from the GeoPHP module "on-the-fly" or by converting the data during insert and update of the Well Known Binary column.

The test environment consisted of a XAMPP stack using XAMPP 1.8.2 with Apache 2.4.3 (Win32 bit version), PHP 5.4.7 and using the MySQL Community Server version 5.7.3.

PHP 5.4.28 is the current stable PHP release version, as of May 2014, as well as MySQL 5.6.7. Many servers, such as the popular Red Hat installation, use PHP 5.3.X and MySQL 5.1 versions, both of which are in their end-of-life periods. MySQL 5.6 also made significant performance enhancements over MySQL 5.1 and 5.5 versions. For these timing tests, MySQL 5.7 was chosen because, even though it is not officially released, the supported geospatial functions like ST_Contains and ST_Within were required for these tests and the database was stable during testing.

The computer used for testing is a Lenovo ThinkPad Edge Laptop, with an AMD A-4300M CPU Radeon(tm) HD Graphics 2.5GHz and 4GB RAM running windows 7 Professional Service Pack 1, 64-Bit.

Due to the fact that a Windows OS was used, the PHP GEOS library was not available. There is only a compiled *nix version. The author of the library, Patrick Hayes, ran tests to compare the performance of GEOS. [1] He concluded that the GEOS library significantly increased geospatial computations. In many cases, tests ran between 30-50% faster using GEOS. To make a significant improvment, the geospatial column should make a 50% speed increase over the existing technologies.

PHP was configured using a memory limit of 512 MB and a post_max_size of 128MB and includes 2 optional libraries: Phar and XMLWriter. These libraries are required due to other web services running on the same machine. Otherwise PHP had been configured using the default settings.

MySQL was also set up using the out-of-the-box installation settings except that table caching was turned off. This was done in order to simulate an environment where database requests would be treated as new queries. Drupal 7's native caching was also turned off to simulate the same on the CMS side.

In addition to Drupal core, the GeoPHP, Views and Chaos Tools (as a required support module) modules were installed and configured. The GeoField 2.x-dev (development) version was installed due to its being a relatively stable version in use by more than 20,000 production Drupal websites, as of March 2014, [2] and supporting modules required a few patches not available in the release version from November 2013.

Two custom data types, "location" and "region" were created and each received a custom GeoField field instance. The "location" data type received the field called "geographic_middle", which was used to store POINT data only. The "region" data type receved the custom GeoField field called "geographic_area" which stored POLYGON data. Doing so created two main tables in the database used to store the data:

```
CREATE TABLE `field_data_field_geographic_middle` (
  `entity_type` varchar(128) NOT NULL DEFAULT '' COMMENT 'The entity
     type this data is attached to',
  `bundle` varchar(128) NOT NULL DEFAULT '' COMMENT 'The field
     instance bundle to which this row belongs, used when deleting a
     field instance',
```

[1] phayes / geoPHP on GitHub
[2] GeoField Module usage statistics

```
  `deleted` tinyint(4) NOT NULL DEFAULT '0' COMMENT 'A boolean
     indicating whether this data item has been deleted',
  `entity_id` int(10) unsigned NOT NULL COMMENT 'The entity id this
     data is attached to',
  `revision_id` int(10) unsigned DEFAULT NULL COMMENT 'The entity
     revision id this data is attached to, or NULL if the entity
     type is not versioned',
  `language` varchar(32) NOT NULL DEFAULT '' COMMENT 'The language
     for this data item.',
  `delta` int(10) unsigned NOT NULL COMMENT 'The sequence number for
     this data item, used for multi-value fields',
  `field_geographic_middle_geom` longblob,
  `field_geographic_middle_geo_type` varchar(64) DEFAULT '',
  `field_geographic_middle_lat` decimal(18,12) DEFAULT NULL,
  `field_geographic_middle_lon` decimal(18,12) DEFAULT NULL,
  `field_geographic_middle_left` decimal(18,12) DEFAULT NULL,
  `field_geographic_middle_top` decimal(18,12) DEFAULT NULL,
  `field_geographic_middle_right` decimal(18,12) DEFAULT NULL,
  `field_geographic_middle_bottom` decimal(18,12) DEFAULT NULL,
  `field_geographic_middle_geohash` varchar(16) DEFAULT NULL,
  PRIMARY KEY
     (`entity_type`,`entity_id`,`deleted`,`delta`,`language`),
  KEY `entity_type` (`entity_type`),
  KEY `bundle` (`bundle`),
  KEY `deleted` (`deleted`),
  KEY `entity_id` (`entity_id`),
  KEY `revision_id` (`revision_id`),
  KEY `language` (`language`),
  KEY `field_geographic_middle_lat` (`field_geographic_middle_lat`),
  KEY `field_geographic_middle_lon` (`field_geographic_middle_lon`),
  KEY `field_geographic_middle_top` (`field_geographic_middle_top`),
  KEY `field_geographic_middle_bottom`
     (`field_geographic_middle_bottom`),
  KEY `field_geographic_middle_left` (`field_geographic_middle_left`),
  KEY `field_geographic_middle_right`
     (`field_geographic_middle_right`),
  KEY `field_geographic_middle_geohash`
     (`field_geographic_middle_geohash`),
  KEY `field_geographic_middle_centroid`
     (`field_geographic_middle_lat`,`field_geographic_middle_lon`),
  KEY `field_geographic_middle_bbox`
     (`field_geographic_middle_top`,`field_geographic_middle_bottom`,`field_geographic_m
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='Data storage for field
    5 (field_geographic_middle)';
```

Both tables were exported by hand and copied to two new tables. They were called field_data_field_geographic_middl and field_data_field_geographic_area_geospatial. A custom script imported the data from an existing website repository that held 6091 location data sets. A second custom script used the

GeoPHP library and the Drupal Hook system to copy data from the Drupal GeoField field instance tables to the custom database tables. The nine provinces of Austria were then also added by hand as region data objects and the polygons copied to the geospatial tables using a second script.

Several timing scripts were then set up to compare the efficiency between GeoPHP, using GeomFromWKB to translate the WKB column to a geospatial data item, and a true geospatial column for the following queries:

1. Obtain the Entity ID numbers of all location data using ST_Within for a given Austrian province.

2. Obtain the Entity ID numbers of all location data using ST_Contains for a given Austrian province. The algorithm is theoretically the same as ST_Within, except that the parameters are reversed.

3. Obtain the Entity ID numbers of all location data using MBRContains (the minimum bounding rectangle) for a given Austrian province.

4. Obtain the computed AREA of a given Austrian province.

5. Obtain the computed distance between location data sets and a given city center POINT (Optimization of the Haversine formula).

In the dataset, the Well Known Text polygon that represents the Province of Vienna is a very simple Polygon object consisting of 21 POINTS. Figure 5.1 outlines the complexity of the polygons that represent the Austrian Provinces. Vienna looks like the following:

```
POLYGON((16.22941971 48.13418961, 16.19574928 48.17251968,
   16.19584084
48.17501068, 16.19868088 48.18561172, 16.21545982 48.23471832,
   16.23734093
48.26472855, 16.39289093 48.33322906, 16.40840912 48.33300018,
   16.42695045
48.33089828, 16.44787979 48.32786179, 16.51655006 48.30213165,
   16.55792999
48.27545929, 16.58342934 48.17860031, 16.58624077 48.15364838,
   16.52095032
48.15169907, 16.44296074 48.15108109, 16.33415031 48.14775848,
   16.27150917
48.1428299, 16.25661087 48.14125061, 16.23889923 48.13835144,
   16.22941971
48.13418961));
```

The test for ST_Within used the following 2 query variations to do timing tests. Each query was run 200 times so an average time per query could be computed. The start and end times were recorded using a custom PHP script. The PHP precision for microtime was set to 12 decimal places.

Province (Number POINTS)	Number of points in POLYGON	Number of Locations returned from ST_Contains	Number of Locations returned from MBRContains
Vienna	21	1424	1531
Lower Austria	129 (108 exterior ring; 21 interior ring)	1332	3085
Upper Austria	88	660	1136
Salzburg	191	527	848
Total from data set	-	6091	6091

Table 5.1: Data set outline

It has been shown on older Windows machines using older versions of PHP that the OS precision is as low as one decimal place. However the results of the timing tests performed here were consistent such that values of up to four decimal places of precision could be gathered. The tests were set up in a manner in which the timing results differed enough that an error range of +/- 0.1 did not affect the conclusions that can be drawn from the results.

The following custom queries were created for testing using the Drupal MySQL PDO:

```
<?php

//Query over the database using the native Drupal GeoField databases
   and MySQL
//PDO
$query = db_select('field_data_field_geographic_middle', 'f');
$query->join('field_data_field_geographic_area', 'f2',
'ST_Within(GeomFromWKB(field_geographic_middle_geom),
GeomFromWKB(field_geographic_area_geom)) = 1');
$query->fields('f', array('entity_id'));
$query->where('f2.entity_id = ' . $nid_of_polygon);
$query->orderBy('f.entity_id', 'ASC');

//SQL Query String:
// SELECT
//   f.entity_id AS entity_id
// FROM
//   field_data_field_geographic_middle f
//   INNER JOIN field_data_field_geographic_area f2
//   ON ST_Within(GeomFromWKB(field_geographic_middle_geom),
//   GeomFromWKB(field_geographic_area_geom)) = 1
// WHERE
//   (f2.entity_id = 19885)
// ORDER BY
//   f.entity_id ASC
```

```
//The same query using a GEOSPATIAL FIELD
$query = db_select('field_data_field_geographic_middle_geospatial',
    'f');
$query->join('field_data_field_geographic_area_geospatial', 'f2',
'ST_Within(field_geographic_middle_geom, field_geographic_area_geom)
    = 1');
$query->fields('f', array('entity_id'));
$query->where('f2.entity_id = ' .
$nid_of_polygon); $query->orderBy('f.entity_id', 'ASC');

//SQL Query String:
// SELECT
//   f.entity_id AS entity_id
// FROM
//   field_data_field_geographic_middle_geospatial f
//   INNER JOIN field_data_field_geographic_area_geospatial f2
//   ON ST_Within(field_geographic_middle_geom,
    field_geographic_area_geom) = 1
// WHERE
//   (f2.entity_id = 19885)
// ORDER BY
//   f.entity_id ASC
```

The GeoPHP library does not currently implement the ST_Within and ST_Contains functions, and does not support the MBRWithin and MBRContains functions in the GeoPHP class objects. The GeoField module does, however, provide a Views API plugin that mimics the MBRWithin function of geospatial columns and makes it readily available to website administrators. The GeoField module stores the top, bottom, left and right values of latitude and longitude in the database for a Polygon object and adds an index over these columns (see table output above) in order to simulate MBRContains and MBRWithin. The adapted geospatial queries were then modeled after the Views API plugin. The Views API plugin provided by GeoPHP does a more complex version of the following query, but the query complexity was reduced. This was done in order to time test the exact lookup and remove any outside influences that could affect the timing of the queries:

```
<?php

$query = db_select('field_data_field_geographic_middle_geospatial',
    'f');
$query->join('field_data_field_geographic_area_geospatial', 'f2',
'f.field_geographic_middle_lat < f2.field_geographic_area_top AND
f.field_geographic_middle_lat > f2.field_geographic_area_bottom AND
f.field_geographic_middle_lon < f2.field_geographic_area_right AND
f.field_geographic_middle_lon > f2.field_geographic_area_left');
$query->fields('f', array('entity_id')); $query->where('f2.entity_id
    = ' .
$nid_of_polygon);
```

```
$query->orderBy('f.entity_id', 'ASC');

// SQL Query String:
// SELECT
//   f.entity_id AS entity_id
// FROM
//   field_data_field_geographic_middle_geospatial f
//   INNER JOIN field_data_field_geographic_area_geospatial f2 ON
//   f.field_geographic_middle_lat < f2.field_geographic_area_top AND
//   f.field_geographic_middle_lat > f2.field_geographic_area_bottom
    AND
//   f.field_geographic_middle_lon < f2.field_geographic_area_right
    AND
//   f.field_geographic_middle_lon > f2.field_geographic_area_left
// WHERE
//   (f2.entity_id = 19885)
// ORDER BY
//   f.entity_id ASC
```

5.2 Hypotheses and expected results

The tests were designed to measure if the performance using Geospatial columns in MySQL improved query lookup times by more than 50% as compared to the GeoPHP library lookup methods, and to the method of converting the Well Known Binary blob data to geospatial column data on-the-fly.

The Area function in PHP does 2n + 2 calculations. The MySQL geospatial function for the area of a polygon in a geospatial field had been optimized and works in a different fashion. Therefore the only hypothesis that we could make is that the area formula in MySQL would be more efficient.

Concerning the Haversine Great-Circle Distance Formula, it was expected that the query would improve by a factor of 6n calculations by saving the radian values to the database instead of the decimal values. We hypothesized that saving the cosine and sine values in the database would only increase performance by 2n calculations becuase the formula still had to calculate (LatA-LatB) and (LonA-LonB) at run time. It was expected, though, that the Haversine Formula would also be made significanly more efficient.

5.3 Results

Tables 5.2 and 5.3 illustrate the average time of the timing tests, the standard deviations, standard errors and finally the 99% confidence range of the timing tests. For ST_ functions, a GeoPHP function equivalent did not exist. Graphs 5.1 and 5.3 show the difference between the Geom-FromText and Geospatial columns. Graphs 5.2 and 5.4 illustrate the relationship of number of points in a polygon versus the response time of the mysql select statement.

Province (Number POINTS)	Average Time Geom-FromWKB	Average Time Geospatial Column	Std. Deviation Geom-FromWKB	Std. Deviation Geospatial Column
Vienna (21)	9.1112	0.5686	0.2820	0.0171
Upper Austria (88)	29.3730	0.3074	6.1289	0.0376
Lower Austria (129)	46.0025	0.7878	5.5205	0.0595
Salzburg (191)	49.4226	0.2472	1.4997	0.0660

Province (Number POINTS)	Std. Error Geom-FromWKB	Std. Error Geospatial Column	99 Percent Confidence Geom-FromWKB	99 Percent Confidence Geospatial Column
Vienna (21)	0.0892	0.0054	0.5136	0.0311
Upper Austria (88)	1.9381	0.0119	11.1630	0.0684
Lower Austria (129)	1.7457	0.0188	10.0550	0.1084
Salzburg (191)	0.4743	0.0209	2.7316	0.1201

Table 5.2: Data for ST_Contains Timing Tests

Table 5.4 displays the data for the minimum bounding regions timing tests, as does the graph 5.5. Table 5.5 and graph 5.6 display the same for the AREA functions.

The same figures are displayed for the various forms of the Haversine formula timing tests performed using the cities of Vienna and Salzburg as the centers of location.

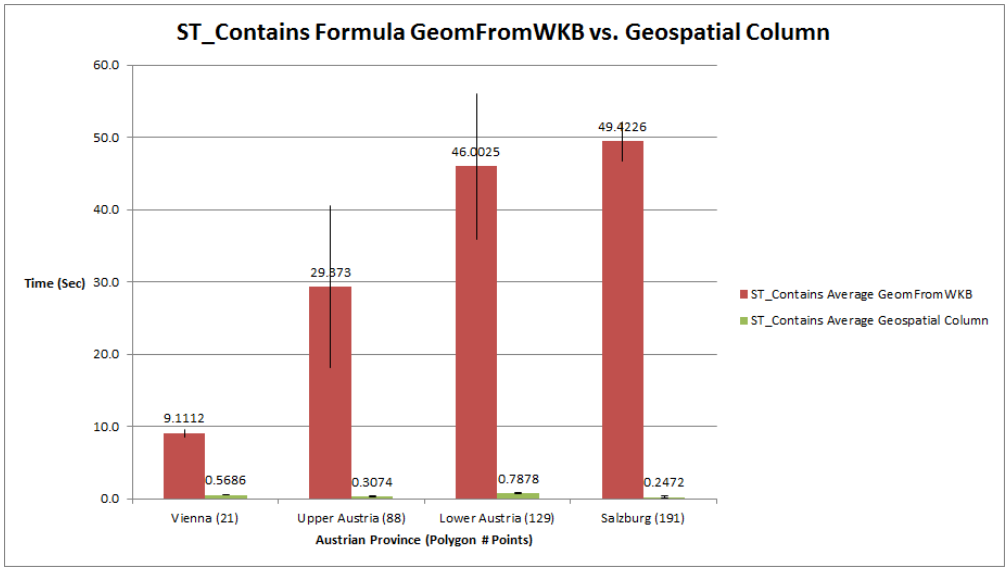

Figure 5.1: ST_Contains timing results

Province (Number POINTS)	**Average Time GeomFromWKB**	**Average Time Geospatial Column**	**Std. Deviation GeomFromWKB**	**Std. Deviation Geospatial Column**
Vienna (21)	11.7172	0.8483	1.6389	0.0410
Upper Austria (88)	25.2927	0.3679	0.4592	0.0174
Lower Austria (129)	33.4597	0.5846	4.5063	0.1124
Salzburg (191)	54.7054	0.3045	5.2248	0.1030

Province (Number POINTS)	**Std. Error GeomFromWKB**	**Std. Error Geospatial Column**	**99 Percent Confidence GeomFromWKB**	**99 Percent Confidence Geospatial Column**
Vienna (21)	0.5183	0.0130	2.9851	0.0746
Upper Austria (88)	0.1452	0.0055	0.8364	0.0317
Lower Austria (129)	1.4250	0.0355	8.2078	0.2046
Salzburg (191)	1.6522	0.0326	9.5164	0.1877

Table 5.3: Data for ST_Within Timing Tests

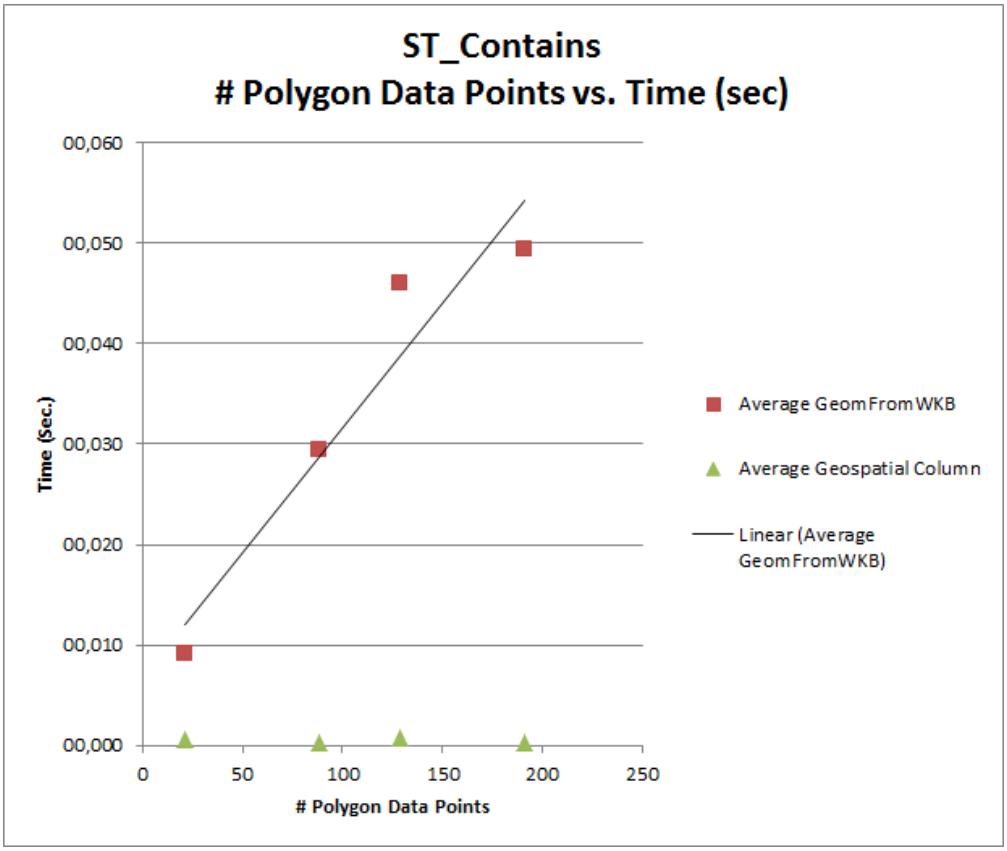

Figure 5.2: ST_Contains timing results in relation to the number of POINTS of each POLYGON

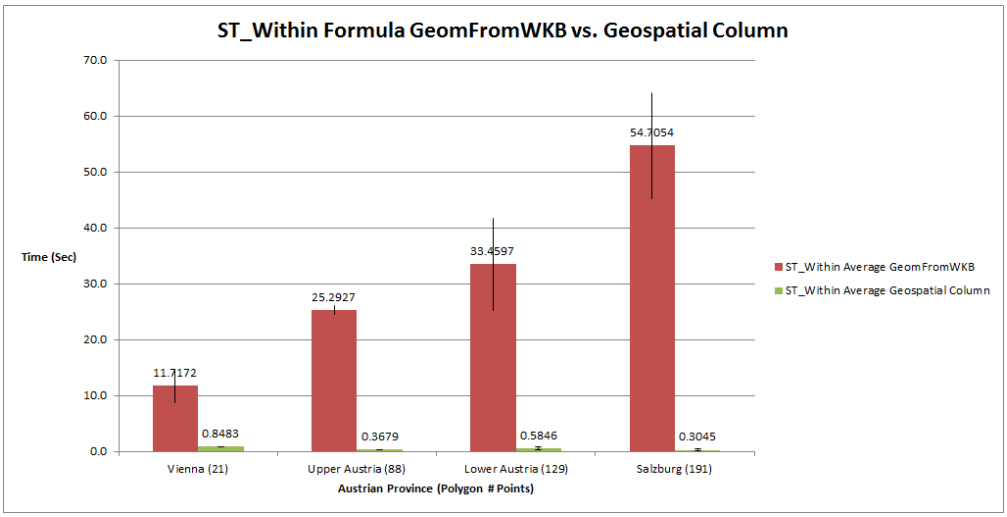

Figure 5.3: ST_Within timing results

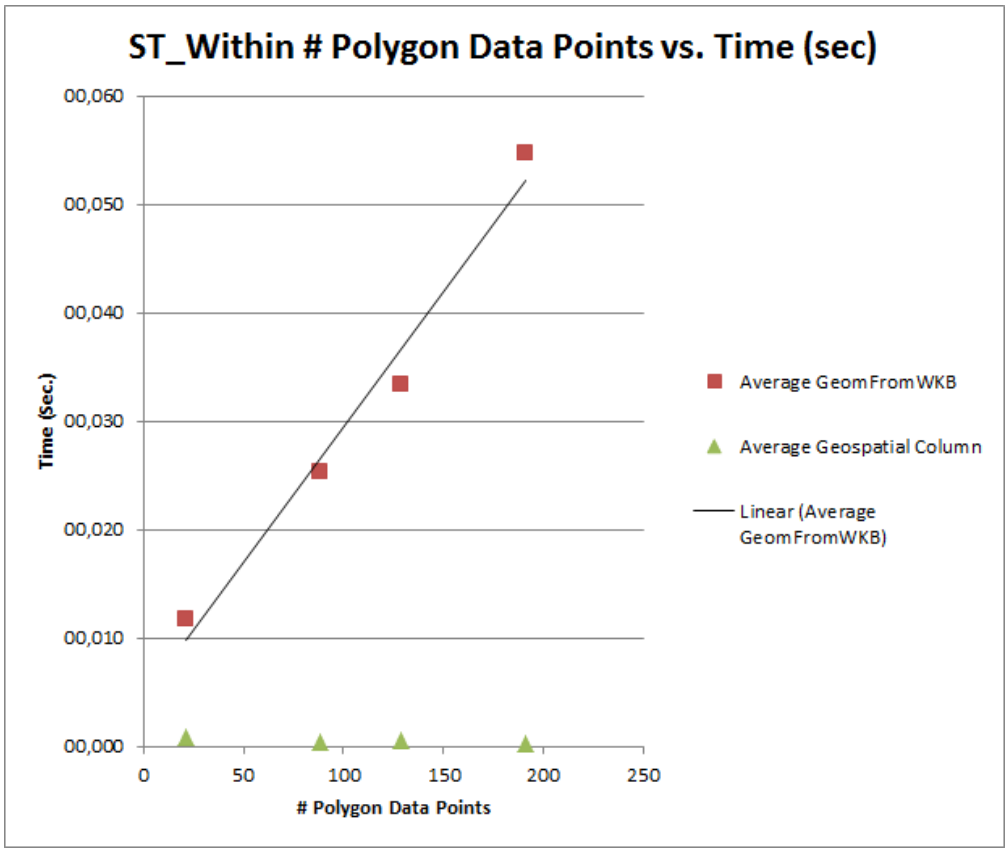

Figure 5.4: ST_Within timing results in relation to the number of POINTS of each POLYGON

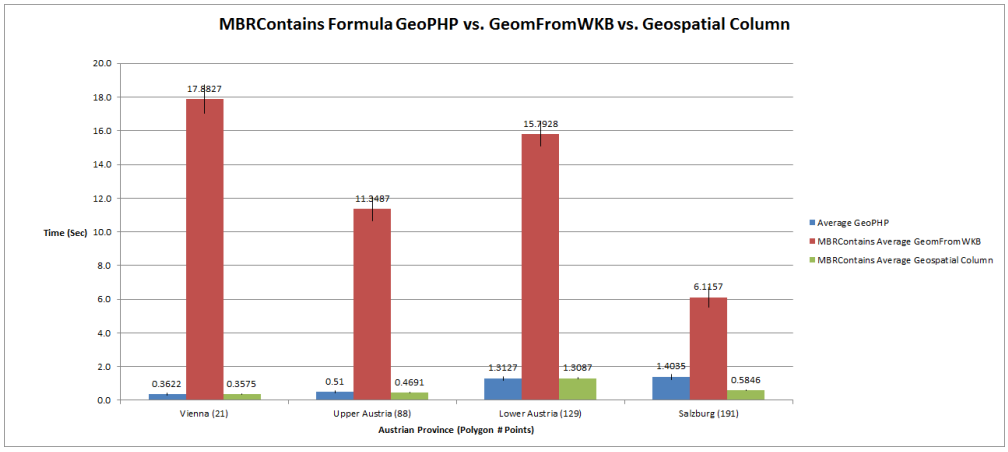

Figure 5.5: MBRContains timing results

Province (Number POINTS)	**Average Time GeoPHP**	**Average Time GeomFromWKB**	**Average Time Geospatial Column**
Vienna (21)	0.3622	17.8827	0.3575
Upper Austria (88)	0.5100	11.3487	0.4691
Lower Austria (129)	1.3127	15.7928	1.3087
Salzburg (191)	1.4035	6.1157	0.5846
Province (Number POINTS)	**Std. Deviation GeoPHP**	**Std. Deviation GeomFromWKB**	**Std. Deviation Geospatial Column**
Vienna (21)	0.0318	0.4692	0.0143
Upper Austria (88)	0.0443	0.3941	0.0150
Lower Austria (129)	0.0635	0.3931	0.0337
Salzburg (191)	0.0877	0.3258	0.0142
Province (Number POINTS)	**Std. Error GeoPHP**	**Std. Error GeomFromWKB**	**Std. Error Geospatial Column**
Vienna (21)	0.0100	0.1484	0.0045
Upper Austria (88)	0.0140	0.1246	0.0048
Lower Austria (129)	0.0201	0.1243	0.0107
Salzburg (191)	0.0277	0.1030	0.0045
Province (Number POINTS)	**99 Percent Confidence GeoPHP**	**99 Percent Confidence GeomFromWKB**	**99 Percent Confidence Geospatial Column**
Vienna (21)	0.0578	0.8545	0.0261
Upper Austria (88)	0.0808	0.7178	0.0274
Lower Austria (129)	0.1156	0.7159	0.0614
Salzburg (191)	0.1598	0.5933	0.0259

Table 5.4: Data for MBRContains Timing Tests

Province (Number POINTS)	**Average Time GeoPHP**	**Average Time GeomFromWKB**	**Average Time Geospatial Column**
Vienna (21)	1.8687	2.0336	1.9196
Upper Austria (88)	6.6674	2.1288	1.8967
Lower Austria (129)	14.4189	2.0128	1.9862
Salzburg (191)	15.9477	3.3725	3.2764

Province (Number POINTS)	**Std. Deviation GeoPHP**	**Std. Deviation GeomFromWKB**	**Std. Deviation Geospatial Column**
Vienna (21)	0.1477	0.1223	0.1246
Upper Austria (88)	0.4123	0.1457	0.0370
Lower Austria (129)	0.3424	0.0393	0.1130
Salzburg (191)	2.7348	0.3254	0.2863

Province (Number POINTS)	**Std. Error GeoPHP**	**Std. Error GeomFromWKB**	**Std. Error Geospatial Column**
Vienna (21)	0.0467	0.0387	0.0394
Upper Austria (88)	0.1304	0.0461	0.0117
Lower Austria (129)	0.1083	0.0124	0.0357
Salzburg (191)	0.8648	0.1029	0.0906

Province (Number POINTS)	**99 Percent Confidence GeoPHP**	**99 Percent Confidence GeomFromWKB**	**99 Percent Confidence Geospatial Column**
Vienna (21)	0.2690	0.2227	0.2269
Upper Austria (88)	0.7510	0.2654	0.0674
Lower Austria (129)	0.6236	0.0716	0.2059
Salzburg (191)	4.9812	0.5927	0.5215

Table 5.5: Data for Area Timing Tests

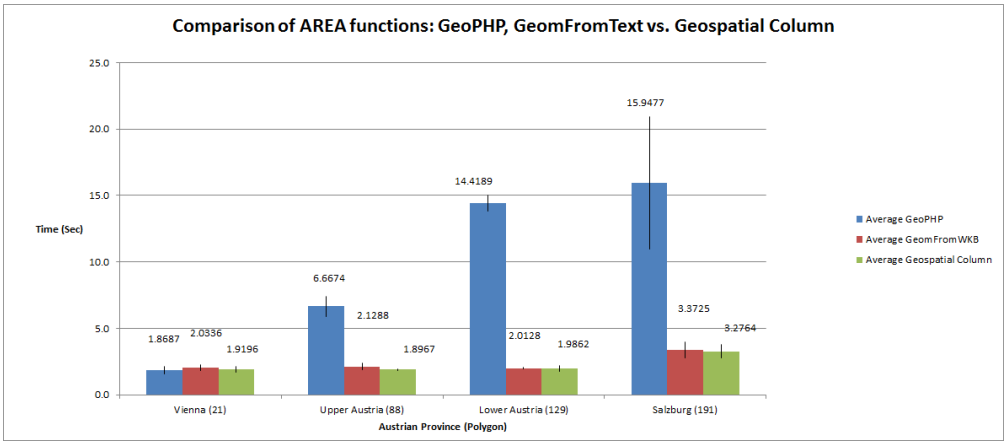

Figure 5.6: AREA function timing results

City (POINT Center)	**Average Decimal**	**Average Radians**	**Average Cosine & Sine**
Vienna (16.22 48.12)	0.0160	0.0141	0.0143
Salzburg (13.33 47.80)	0.0146	0.0144	0.0137
Province (Number POINTS)	**Std. Deviation GeoPHP**	**Std. Deviation GeomFromWKB**	**Std. Deviation Geospatial Column**
Vienna (16.22 48.12)	0.0020	0.0009	0.0009
Salzburg (13.33 47.80)	0.0014	0.0014	0.0006
Province (Number POINTS)	**Std. Error GeoPHP**	**Std. Error GeomFromWKB**	**Std. Error Geospatial Column**
Vienna (16.22 48.12)	0.0005	0.0002	0.0002
Salzburg (13.33 47.80)	0.0004	0.0003	0.0002
Province (Number POINTS)	**99 Percent Confidence GeoPHP**	**99 Percent Confidence GeomFromWKB**	**99 Percent Confidence Geospatial Column**
Vienna (16.22 48.12)	0.0013	0.0006	0.0006
Salzburg (13.33 47.80)	0.0009	0.0009	0.0004

Table 5.6: Data for Area Timing Tests

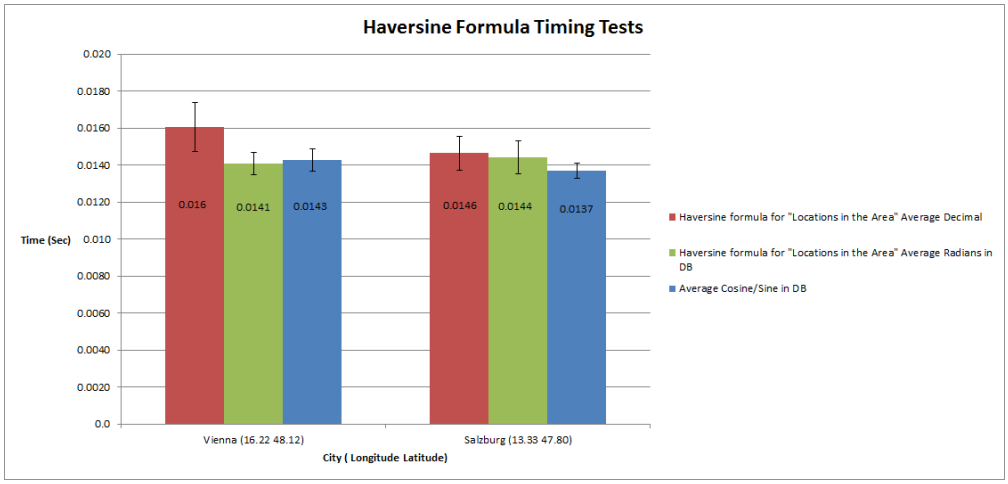

Figure 5.7: Computed great circle distance from a fixed point using the Haversine Formula

CHAPTER 6

Discussion of Results, Conclusions and Outlook

6.1 ST_Contains & ST_Within

As is illustrated in figures 5.2 and 5.4, the number of data points in the polygon influences the speed of the mysql query. The more detailed and accurate the polygon is, the slower the query. Such a relationship did not exist in the case of the geospatial column solution, indicating that the geospatial column query performance is not directly related to the complexity of the data alone.

The test data supports the hypothesis that a geospatial column is signficantly faster. Querying the data using a geospatial column brings a performance increase of 1600% in the case of the Province of Vienna, with more complex POLYGON'S showing a higher increase in performance.

6.2 MBRContains

Here the data from table 5.4 shows mixed results. A query that uses the GeoPHP library querying the four floating point mysql columns that hold the data for "top", "bottom", "left" and "right" showed that the query was slightly slower than querying against a geospatial column. The margin of difference was not large enough to be significant, as the 99% confidence levels for the provinces of Vienna and Upper Austria show that a query against the four floating point columns in the table was still, at times, faster than the geospatial columns query. In theory, since the GeoPHP query uses the four floating point numbers in the database, there should not be a direct relationship of polygon complexity and query efficiency, which can be seen in graph 5.4. A geospatial column, as opposed to the four floating point columns, still saves data space and reduces query lookup complexity. It is advisable to use the geospatial column solution where possible.

The hypothesis that querying over a geospatial column is significantly faster than querying against a Well Known Binary blob column and translating the data to geospatial data on-the-fly was supported. The data was unable to show, however, that a relationship between the number of polygon data points and the query performance exist. Querying for location data in the province of Upper Austria, for example, a polygon of 88 data points, was faster than the same query for location data within the province of Vienna, a polygon of only 21 data points. Lower Austria, 129 data points, was slower than both Upper Austria and Salzburg, where Salzburg had 191 data points.

6.3 AREA

As figure 5.6 shows, determining the area of a polygon using the GeoPHP library is proportional to the number of data points in the polygon. It is also significantly slower than both the geospatial column and the GeomFromText query methods. The difference between the lookup using GeomFromText on a Well Known Binary column and the geospatial column was, however, not significant. The hypothesis that the Area function is significantly faster for a geospatial column compared to a non-geospatial column was rejected.

6.4 Distance from a point using the Haversine Formula

The Haversine distance formula can be marginally increased in speed if the decimal coordinate values are converted to radian values before being saved to the database. The slowest average calculation of the Haversine formula without database optimization was 0.016 seconds, with Vienna as the center of location, as opposed to an optimized database that could run the query at 0.0143 seconds. The 99% confidence interval is also large enough that we could only be certain of an increase in performance due to the conversion to radians. Saving the Cosine and Sine values to the database did not yield any measurable performance improvement.

The Haversine formula queries however, are significantly less time consuming, in general, than the ST_Within, ST_Contains, MBRContains and the Area functions that were tested. Optimization of the database for the geospatial column has a higher priority if a system is using those functions, rather than optimization of the Haversine formula.

6.5 Conclusion of research questions

This paper has set out to discuss the following questions:

- How should geo data be structured by web-based CMSs so that the system, and the administrator, can efficiently and flexibly manage the geo data and display it in context?
  - What data, if any, should be stored locally?
  - In what ways should the geo data be structured?

- What external sources of geo data are currently available and how can these sources be efficiently used or imported into the CMS?

By evaluating existing theories and models, such as the OpenGIS Geometry Model, we were able to see how web-based CMSs should structure the data locally. It was shown by way of prototyping real-world data that it is advisable to optimize the systems. This would take advantage of geospatial columns on the database level for storing geospatial data so that it is accessible in a flexible manner. Doing so increases the ability to evaluate properties and relationships between geospatial objects.

The use-cases that were identified and discussed included managing geospatial data locally in the web-based CMS and displaying that data in various formats, such as on the web-page or in a static or slippy map. We discussed geocoding address information into latitude and longitude as well as reverse-geocoding. Also, there are many formats to print geospatial data as a file, inclduing GeoJSON, KML and GML. These file formats can be used to share the geospatial data with other sites and programs, such as Google Earth.

Far more complex computations include determing properties of geospatial data, such as the length of a Linestring or the area of a Polygon. Also, determing relationships between geospatial features, such as the distance between two points, or grouping points using geoclustering to improve usability, can be done using existing technologies.

External geospatial data sources are also easily accessed, such as the Open Government Data project and geocoder services provided by Google or Yahoo. We often concentrated on the Drupal CMS since it has the APIs neccessary to support exchanging information between these repositories. It does this by importing and printing geospatial file formats using GeoRSS, glsgeojson, glsrest, glsgml and glskml file formats.

Data from these items can be imported into slippy mapping systems such as Google Maps API v3, Openlayers and Leaflet, or imported into other mapping systems such as Google's Static Maps API and Google's Google Earth software. There is also the ArcGIS software which can import shapefiles.

Database engines like MySQL and PostGIS will continue to refine their geospatial support to make lookup times faster and reduce energy expense. The future of geospatial data in websites will be to refine the User Interface of its systems and normalize API systems, as well as offering increased flexibility of output. There is still much work to be done for the Open Source web-based CMSs of the future.

APPENDIX A

Appendix A: Index

A.1 Glossary

ATOM: The Atom Syndication Format is a specification for XML files intended as web feeds

FieldAPI: The API in Drupal for interacting with fields

Field display formatter: Field formatters tell how the data of a field in Drupal should be printed

Field input widget: Input widgets tell how the data of a field in Drupal should be gathered

Geo Microformat: The Microformat for displaying geospatial information.

GeoJSON: JSON specification to describe geospatial objects

GeoRSS: RSS specification file format to include geographical point data.

Hook: In Drupal, a hook is a key name that modules and themes can invoke so they are able to respond to events.

Minimum Bounding Region: The smallest box that includes a geospatial feature. Defined by top, right, bottom, left

Node: A node in Drupal is the term used to define a content object. Articles, Pages, etc. are nodes

OpenGIS Geometry Model: The model created by the Open Geospatial Consortium to represent spatial features.

Projection: A projection is the process of displaying a 3D object on a 2D surface

Raster: A type of image that uses a grid consisting of pixels.

Render Array: An array that contains both the content and the information of how the content is to be displayed in Drupal.

RESTful web services: A service that allows actions to be made on a website using a simple HTTP protocol
Slippy Maps: Mapping programs such as Google Maps that use tiles, a viewport and raster items to generate maps

Spatial Reference System ID: An ID for spatial data to differentiate between coordinate systems

Taxonomy: The Drupal term for classifying content. Also referred to as categories

Uniform Resource Identifiers: strings that identifies the location of content, usually on the internet

User Interface: The method in which humans and machines interact. In Web-based applications, a User Interface is the webpage

Well Known Binary: A binary representation of spatial objects for the OpenGIS Geometry Model

Well Known Text: A text-based representation of spatial objects for the OpenGIS Geometry Model

A.2 List of Acronyms

API: Application Programming Interfaces

ATOM: Atom Syndication Format

CSV: Comma separated file

CMS: Content Management System

GIS: Geographical Information Systems

GML: Geography Markup Language

GPS: Global Positioning Satellite

KML: Keyhole Markup Language

JSON: JavaScript Object Notation

MBR: Minimum Bounding Region

MVC: Model-View-Control

OGC: Open Geospatial Consortium

RDFa: Resource Description Framework in Attributes REST: RESTful web services
RSS: Rich site summary or Really Simple Syndication

SRID: Spatial Reference System ID

URI: Uniform Resource Identifiers

UI: User Interface

WKB: Well Known Binary

WKT: Well Known Text

XML: Extensible Markup Language

List of Figures

1.1 The technology stack of the Geolocation and Geomap modules on top of Drupal and its required technologies 2
1.2 MySQL geospatial column data types 6

3.1 Displaying the input POLYGON, known as Well Known Text data, for the provinces of Vienna and Lower Austria on an OpenLayers map. The data is from an open source database, illustrating accuracy of data obtained from third party sources. The polygon for Lower Austria is much more accurate than for Vienna. 28
3.2 The raw data printed to a page using a Drupal View and the GeoField module. . . . 29
3.3 A generated static map jpeg file using the Google Static Maps API and the default settings in Drupal's Geolocation module. 29
3.4 The default behavior in Chrome using the HTML5 Gelocation specification. A map is generated of the world and a blue dot signifies the position returned from the navigator object. 30
3.5 Using the Geolocation module in Drupal for geocoding a string. The input widget is a simple text field. When the button "Get Location" is pressed, an AJAX request is sent to the Google Geocoder Service. The method is more straight foward than the GeoField geocoder solution but is also less flexible. 30
3.6 Driving directions provided by the Google driving directions service for travelling by car from St. Stephen's Cathedral to the Vienna University of Technology. 30
3.7 Without using a clustering method, the data on this map looks cluttered. 31
3.8 A map of the Drupalcamp Vienna attendees using the Geocluster and GeoField modules in Drupal 31

4.1 OGC Geometry Model data types 34
4.2 MySQL Geospatial Column Data Types 38
4.3 Widget types defined in the basic Geolocation module and the additional Geolocation Google Maps module. 42
4.4 Handler types defined in the basic Geolocation module and the additional Geolocation Google Maps module. 42
4.5 Handler types defined in the base Geolocation module and the Geolocation Google Maps module. 43
4.6 Creating a GeoField that will automatically import its data from a KML file field . 46

5.1 ST_Contains timing results 57
5.2 ST_Contains timing results in relation to the number of POINTS of each POLYGON 58
5.3 ST_Within timing results 58
5.4 ST_Within timing results in relation to the number of POINTS of each POLYGON 59
5.5 MBRContains timing results 59
5.6 AREA function timing results 62
5.7 Computed great circle distance from a fixed point using the Haversine Formula . . 63

List of Tables

5.1 Data set outline 53
5.2 Data for ST_Contains Timing Tests 56
5.3 Data for ST_Within Timing Tests 57
5.4 Data for MBRContains Timing Tests 60
5.5 Data for Area Timing Tests 61
5.6 Data for Area Timing Tests 62

APPENDIX B

Appendix B: Timing Tests

B.1 Synchronisation of GeoField Table for Geospatial Column Values

```
<?php

/**
 * @file
 * Sync the geospatial table with the geofield field table instance
 */

/**
 * Root directory of Drupal installation.
 */
define('DRUPAL_ROOT', getcwd());

require_once DRUPAL_ROOT . '/includes/bootstrap.inc';
drupal_bootstrap(DRUPAL_BOOTSTRAP_FULL);

$original_table = 'field_data_field_geographic_middle';
$geospatial_custom_table =
   'field_data_field_geographic_middle_geospatial';

geophp_load();

$select = db_select($original_table, 'f');
$select->fields('f');
$select->range(0, 10000);
$data = $select->execute();
```

```
db_query('TRUNCATE TABLE
    field_data_field_geographic_middle_geospatial')->execute();

foreach ($data as $row) {
   $geom = geoPHP::load($row->field_geographic_middle_geom, 'wkb');
   $row->field_geographic_middle_geom = $geom->out('wkt');

   //translate this info now into the custom table
   $insert =
      db_insert('field_data_field_geographic_middle_geospatial');
   $insert->fields(array(
    'entity_type' => $row->entity_type,
    'bundle' => $row->bundle,
    'deleted' => $row->deleted,
    'entity_id' => $row->entity_id,
    'revision_id' => $row->revision_id,
    'language' => $row->language,
    'delta' => $row->delta,
    'field_geographic_middle_geo_type' =>
       $row->field_geographic_middle_geo_type,
    'field_geographic_middle_lat' =>
       $row->field_geographic_middle_lat,
    'field_geographic_middle_lon' =>
       $row->field_geographic_middle_lon,
    'field_geographic_middle_left' =>
       $row->field_geographic_middle_left,
    'field_geographic_middle_top' =>
       $row->field_geographic_middle_top,
    'field_geographic_middle_right' =>
       $row->field_geographic_middle_right,
    'field_geographic_middle_bottom' =>
       $row->field_geographic_middle_bottom,
    'field_geographic_middle_geohash' =>
       $row->field_geographic_middle_geohash,
   ));

   $insert->execute();

   $update =
      db_update('field_data_field_geographic_middle_geospatial');
   $update->expression('field_geographic_middle_geom',
      "GeomFromText('" . $geom->out('wkt') . "')");
   $update->condition('entity_type', $row->entity_type);
   $update->condition('bundle', $row->bundle);
   $update->condition('deleted', $row->deleted);
   $update->condition('entity_id', $row->entity_id);
   $update->condition('revision_id', $row->revision_id);
   $update->condition('language', $row->language);
   $update->condition('delta', $row->delta);
```

```
    $update->execute();
}
```

B.2 Synchronisation of GeoField Table for Radians, Sine and Cosine

```
<?php
define('DRUPAL_ROOT', getcwd());

require_once DRUPAL_ROOT . '/includes/bootstrap.inc';
drupal_bootstrap(DRUPAL_BOOTSTRAP_FULL);

$original_table = 'field_data_field_geographic_middle';
$geospatial_custom_table =
'field_data_field_geographic_middle_sine_and_cosine';

geophp_load();

$select = db_select($original_table, 'f');
$select->fields('f');
$select->addExpression('RADIANS(field_geographic_middle_lat)',
    'latr');
$select->addExpression('RADIANS(field_geographic_middle_lon)',
    'lonr');
$select->addExpression('SIN(RADIANS(field_geographic_middle_lat))',
    'lats');
$select->addExpression('COS(RADIANS(field_geographic_middle_lat))',
    'latc');
$select->addExpression('SIN(RADIANS(field_geographic_middle_lon))',
    'lons');
$select->addExpression('COS(RADIANS(field_geographic_middle_lon))',
    'lonc');
$select->range(0, 10000);
$data = $select->execute();

db_query('TRUNCATE TABLE ' . $geospatial_custom_table)->execute();

foreach ($data as $row) {

    //translate this info now into the custom table
    $insert = db_insert($geospatial_custom_table);
    $insert->fields(array(
      'entity_type' => $row->entity_type,
      'bundle' => $row->bundle,
      'deleted' => $row->deleted,
```

```
    'entity_id' => $row->entity_id,
    'revision_id' => $row->revision_id,
    'language' => $row->language,
    'delta' => $row->delta,
    'field_geographic_middle_geom' =>
       $row->field_geographic_middle_geom,
    'field_geographic_middle_geo_type' =>
       $row->field_geographic_middle_geo_type,
    'field_geographic_middle_lat_radians' => $row->latr,
    'field_geographic_middle_lon_radians' => $row->lonr,
    'field_geographic_middle_lat_sine' => $row->lats,
    'field_geographic_middle_lat_cosine' => $row->latc,
    'field_geographic_middle_lon_sine' => $row->lons,
    'field_geographic_middle_lon_cosine' => $row->lonc,
    'field_geographic_middle_left' =>
       $row->field_geographic_middle_left,
    'field_geographic_middle_top' =>
       $row->field_geographic_middle_top,
    'field_geographic_middle_right' =>
       $row->field_geographic_middle_right,
    'field_geographic_middle_bottom' =>
       $row->field_geographic_middle_bottom,
    'field_geographic_middle_geohash' =>
       $row->field_geographic_middle_geohash,
  ));

  $insert->execute();
}
```

B.3 Timing Test ST_Within and ST_Contains

Variations of this script were created for several polygon data. The province of Vienna is polygon number 19885.

```
<?php
$microtime_start_of_script = microtime(true);

define('DRUPAL_ROOT', getcwd());

require_once DRUPAL_ROOT . '/includes/bootstrap.inc';
drupal_bootstrap(DRUPAL_BOOTSTRAP_FULL);

drupal_set_title('Timing Vienna ST_Within & ST_Contains.');

geophp_load();

$i = 0;
```

```
$j = 0;

$timer = array('#theme' => 'table', '#header' =>
    array('Measurement', 'start
timestamp', 'end timestamp', 'Measurement duration (ms)', 'Time
    elapsed since
start of script (ms)'),);

$timer['#rows'][$i] = array('Script start to end of Drupal
    initialization
routine.', $microtime_start_of_script, microtime (true));

$timer['#rows'][++$i] = array('Load polygon node:', microtime
    (true));

//the node id with the polygon data, it will be loaded into the
    script
$nid_of_polygon = 19885;
$node_with_polygon = node_load($nid_of_polygon);

$timer['#rows'][$i][2] = microtime (true);
$timer['#rows'][++$i] = array('Assembling queries:', microtime
    (true));

//polygon in WKT
$polygon =
    $node_with_polygon->field_geographic_area[LANGUAGE_NONE][0]['geom'];

//timing test for query of a blob column that needs to convert data
    to a
//GEOSPATIAL DATA TYPE
$query = db_select('field_data_field_geographic_middle', 'f');
$query->join('field_data_field_geographic_area', 'f2',
'ST_Within(GeomFromWKB(field_geographic_middle_geom),
GeomFromWKB(field_geographic_area_geom)) = 1');
$query->fields('f', array('entity_id'));
$query->where('f2.entity_id = ' . $nid_of_polygon);
$query->orderBy('f.entity_id', 'ASC'); $queries[$j++] =
    str_replace(array('{',
'}'), '', $query->__toString());

//timing test for query of a Geospatial column that does not need to
    convert data
$query = db_select('field_data_field_geographic_middle_geospatial',
    'f');
$query->join('field_data_field_geographic_area_geospatial', 'f2',
'ST_Within(field_geographic_middle_geom, field_geographic_area_geom)
    = 1');
$query->fields('f', array('entity_id'));
```

```
$query->where('f2.entity_id = ' . $nid_of_polygon);
$query->orderBy('f.entity_id', 'ASC');
$queries[$j++] = str_replace(array('{', '}'), '',
    $query->__toString());

$timer['#rows'][$i][2] = microtime (true);

foreach ($queries as $k => $query) {
   for ($j = 0; $j < 11; $j++) {

      //repeat 200 times so that we get a good microtime value
      $timer['#rows'][++$i] = array('Query ' . $k . ' timing query
         result ' .
      $j, microtime (true)); for ($m = 0; $m < 200; $m++) {
         $resultset = db_query($query);
      }
      $timer['#rows'][$i][2] = microtime (true);
      $timer['#rows'][$i][0] .= ' (results: ' .
         $resultset->rowCount() . ')';
   }
}

$timer['#rows'][++$i] = array('Create output:', microtime (true));

$content = array(
   array(
      '#markup' => '<p>The following test will query the database
         for all Event Locations both'
         . ' inside and outside of Vienna using a static
            Polygon.<p><fieldset><label>Node: '
         . $node_with_polygon->title . '</label><textarea
            style="width: 90%; height: 300px;">'
         . print_r($node_with_polygon, 1) . '</textarea></fieldset>',
   ),
);

foreach ($queries as $k => $query) {
   $content[] = array(
      '#markup' => '<fieldset><label>Query ' . $k .
         ':</label><textarea
      style="width: 90%; height: 300px;">' . $query .
         '</textarea></fieldset>',
   );
}

$timer['#rows'][$i][2] = microtime (true);
foreach ($timer['#rows'] as $k => $v) {
   $timer['#rows'][$k][3] = (float) ($v[2] - $v[1]);
   $timer['#rows'][$k][4] = (float) ($v[2] -
```

```
      $microtime_start_of_script);
}
$content[] = $timer;

$temp = array();
foreach ($timer['#rows'] as $k => $v) {
   $temp['#markup'][] = str_replace('.', ',', (string) $v[3]);
}
$temp['#markup'] = '<fieldset><label>Query timing:</label><textarea
style="width: 90%; height: 300px;">' . implode("\n",
    $temp['#markup']) .
'</textarea></fieldset>';
$content[] = $temp;

print drupal_render_page(array('content' => $content));
```

B.4 Timing Test MBRContains

```
<?php
$microtime_start_of_script = microtime(true);

define('DRUPAL_ROOT', getcwd());

require_once DRUPAL_ROOT . '/includes/bootstrap.inc';
drupal_bootstrap(DRUPAL_BOOTSTRAP_FULL);

drupal_set_title('Timing: compare MBRWithin queries.');

geophp_load();

$i = 0;
$j = 0;

$timer = array('#theme' => 'table', '#header' =>
   array('Measurement', 'start
timestamp', 'end timestamp', 'Measurement duration (s)', 'Time
   elapsed since
start of script (sec)'),);

$timer['#rows'][$i] = array('Script start to end of Drupal
   initialization
routine.', $microtime_start_of_script, microtime (true));
$timer['#rows'][++$i] = array('Load polygon node:', microtime
   (true));

$nid_of_polygon = 19885;
```

```
$node_with_polygon = node_load($nid_of_polygon);

$timer['#rows'][$i][2] = microtime (true);
$timer['#rows'][++$i] = array('Assembling queries:', microtime
    (true));

//polygon in WKT
$polygon =
    $node_with_polygon->field_geographic_area[LANGUAGE_NONE][0]['geom'];

$query = db_select('field_data_field_geographic_middle', 'f');
$query->join('field_data_field_geographic_area', 'f2',
'MBRContains(GeomFromWKB(field_geographic_area_geom),
GeomFromWKB(field_geographic_middle_geom)) = 1');
$query->fields('f', array('entity_id'));
$query->where('f2.entity_id = ' . $nid_of_polygon);
$query->orderBy('f.entity_id', 'ASC');
$queries[$j++] = str_replace(array('{', '}'), '',
    $query->__toString());

//timing test for query of a Geospatial column that does not need to
    convert data
$query = db_select('field_data_field_geographic_middle_geospatial',
    'f');
$query->join('field_data_field_geographic_area_geospatial', 'f2',
'MBRContains(field_geographic_area_geom,
    field_geographic_middle_geom) = 1');
$query->fields('f', array('entity_id'));
$query->where('f2.entity_id = ' . $nid_of_polygon);
$query->orderBy('f.entity_id', 'ASC');
$queries[$j++] = str_replace(array('{', '}'), '',
    $query->__toString());

$timer['#rows'][$i][2] = microtime (true);

foreach ($queries as $k => $query) {
   for ($j = 0; $j < 11; $j++) {

      //repeat 200 times so that we get a good microtime value
      $timer['#rows'][++$i] = array('Query ' . $k . ' timing query
          result ' .
      $j, microtime (true)); for ($m = 0; $m < 200; $m++) {
         $resultset = db_query($query);
      }
      $timer['#rows'][$i][2] = microtime (true);
      $timer['#rows'][$i][0] .= ' (results: ' .
          $resultset->rowCount() . ')';
   }
}
```

```
$timer['#rows'][++$i] = array('Create output:', microtime (true));

$content = array(
   array(
      '#markup' => '<p>The following test will query the database
         for all Event Locations both'
         . ' inside and outside of Vienna using a static
            Polygon.<p><fieldset><label>Node: '
         . $node_with_polygon->title . '</label><textarea
            style="width: 90%; height: 300px;">'
         . print_r($node_with_polygon, 1) . '</textarea></fieldset>',
   ),
);

foreach ($queries as $k => $query) {
   $content[] = array(
      '#markup' => '<fieldset><label>Query ' . $k .
         ':</label><textarea
      style="width: 90%; height: 300px;">' . $query .
         '</textarea></fieldset>',
   );
}

$timer['#rows'][$i][2] = microtime (true);
foreach ($timer['#rows'] as $k => $v) {
   $timer['#rows'][$k][3] = (float) ($v[2] - $v[1]);
   $timer['#rows'][$k][4] = (float) ($v[2] -
       $microtime_start_of_script);
}
$content[] = $timer;

$temp = array();
foreach ($timer['#rows'] as $k => $v) {
   $temp['#markup'][] = str_replace('.', ',', (string) $v[3]);
}
$temp['#markup'] = '<fieldset><label>Query timing:</label><textarea
style="width: 90%; height: 300px;">' . implode("\n",
    $temp['#markup']) .
'</textarea></fieldset>';
$content[] = $temp;

print drupal_render_page(array('content' => $content));
```

B.5 Haversine Great Circle Distance Formula Timing Tests

```
<?php

$microtime_start_of_script = microtime(true);

define('DRUPAL_ROOT', getcwd());

require_once DRUPAL_ROOT . '/includes/bootstrap.inc';
drupal_bootstrap(DRUPAL_BOOTSTRAP_FULL);

$queries = array();

$content = array();

geophp_load();

$i = 0;
$j = 0;

$timer = array('#theme' => 'table', '#header' =>
    array('Measurement', 'start
timestamp', 'end timestamp', 'Measurement duration (ms)', 'Time
    elapsed since
start of script (ms)'),);

$timer['#rows'][$i] = array('Script start to end of Drupal
    initialization
routine.', $microtime_start_of_script, microtime (true));
$timer['#rows'][++$i] = array('Load polygon node:', microtime
    (true));

//the node id with the polygon data, it will be loaded into the
    script
$node_with_polygon = node_load(19885);

$timer['#rows'][$i][2] = microtime (true);
$timer['#rows'][++$i] = array('Assembling queries:', microtime
    (true));

//timing test for query - state of the art
$queries[] = "SELECT f.entity_id, ( 6371 * ACOS( COS(
RADIANS(48.12) ) * f.field_geographic_middle_lat_cosine * COS(
f.field_geographic_middle_lon_radians - RADIANS(16.22) ) + SIN(
    RADIANS(48.12) )
* f.field_geographic_middle_lat_sine ) ) AS distance FROM
field_data_field_geographic_middle_sine_and_cosine f WHERE 1 ORDER
    BY distance
ASC";

//timing test where data is already in RADIANS
```

```
$queries[] = "SELECT f.entity_id, ( 6371 * ACOS( COS ( 0.839427 ) *
   COS (
f.field_geographic_middle_lat_radians ) * COS (
f.field_geographic_middle_lon_radians - 0.282949 ) + SIN (
0.83942 ) * SIN ( f.field_geographic_middle_lat_radians ) ) ) AS
   distance FROM
field_data_field_geographic_middle_radians f WHERE 1 ORDER BY
   distance ASC";

//timing test where cosine & sine data is already available
$queries[] = "SELECT f.entity_id, ( 6371 * ACOS( 0,999 ) *
f.field_geographic_middle_lat_cosine * COS(
f.field_geographic_middle_lon_radians - 0.2830 ) + 0.0147 ) *
f.field_geographic_middle_lat_sine ) ) AS distance FROM
field_data_field_geographic_middle_sine_and_cosine f WHERE 1 ORDER
   BY distance
ASC";

foreach ($queries as $query) {
   $timer['#rows'][$i][2] = microtime (true);
   $timer['#rows'][++$i] = array('Execute query:', microtime (true));
   $resultset = db_query($query);
   $timer['#rows'][$i][2] = microtime (true);
   $timer['#rows'][++$i] = array('Print Results:', microtime (true));
}

$timer['#rows'][$i][2] = microtime (true);
$timer['#rows'][++$i] = array('Create output:', microtime (true));

foreach ($queries as $k => $query) {
   $content[] = array(
      '#markup' => '<fieldset><label>Query ' . $k .
         ':</label><textarea
      style="width: 90%; height: 300px;">' . $query .
         '</textarea></fieldset>',
   );
}

$timer['#rows'][$i][2] = microtime (true);
foreach ($timer['#rows'] as $k => $v) {
   $timer['#rows'][$k][3] = (float) ($v[2] - $v[1]);
   $timer['#rows'][$k][4] = (float) ($v[2] -
      $microtime_start_of_script);
}
$content[] = $timer;
print drupal_render_page(array('content' => $content));
```

Bibliography

[-98] -. *ESRI Shapefile Technical Description*. Environmental Systems Research Institute, Inc., 1 edition, July 1998.

[-08] -. Global positioning system standard positioning service performance standard. Technical Report 4, Department of Defense United States of America; GPS Navstar Global Positioning System, September 2008.

[14] *MySQL 5.7 Reference Manual*. Oracle Corporation, 2014. http://dev.mysql.com/doc/refman/5.7/en/gis-data-formats.html.

[Bis] Bishr, Yaser PhD. What is your context? http://www.ncgia.ucsb.edu/projects/nga/docs/Bishr_Position.pdf, Accessed: 2013-02-11.

[Biz09] Bizer, Christian; Heath, Tom; Berners-Lee, Tim. Linked data - the story so far, 2009. Preprint, http://tomheath.com/papers/bizer-heath-berners-lee-ijswis-linked-data.pdf.

[Dab13] Dabernig, Joseph. Geocluster: Server-side clustering for mapping in drupal based on geohash. Master's thesis, Vienna University of Technology, June 2013.

[Her10] Herring, John R. *OpenGIS® Implementation Standard for Geographic information - Simple feature access - Part 2: SQL option*. Open Geospatial Consortium Inc., August 2010. Reference Number: OGC 06-104r4;.

[Hol11] Holdener III, Anthony T. *HTML5 Geolocation: Bringing Location to Web Applications*. O'Reilly Media, May 2011.

[Mee06] Meert, Wannes. Clustering maps. Master's thesis, Katholieke Universitaet Leuven, 2006.

[Mri08] Mrissa, Michael; Al-Jabari, Mohanad; Thiran, Philippe. Using microformats to personalize web experience. In *ICWE 2008 Workshops, 7th Int. Workshop on Web-Oriented Software Technologies – IWWOST 2008*, pages 63–68, Bratislava, Slovakia, July 2008. Vydavateľstvo STU.

[Pal07] Palazzolo, Alan; Turnbull, Thomas. *Mapping with Drupal*. O'Reilly Media, 3 edition, 2007.

[Rax11] Rax, Suprio; Simion, Bogdan; Brown, Angela Demke. Jackpine: a benchmarkt to evaluate spatial database performance. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1139–1150. IEEE, April 2011.

[Ree73] Reenskaug, Trygve. *Administrative Control in the Shipyard*, August 1973. Accessed 20.12.2013, http://heim.ifi.uio.no/ trygver/themes/mvc/mvc-index.html.

[Rub06] Rubin, Alexander. Geo/spatial search with mysql, 2006. http://de.scribd.com/doc/2569355/Geo-Distance-Search-with-MySQL; Accessed: 2013-02-07; Posted by Kovyrin, Oleksiy.

[San08] Sandvik, Bjørn. Using kml for thematic mapping. MSc in Geographical Information Science 2008, August 2008.

[Ude08] Udell, Sterling. *Beginning Google Maps Mashups with Mapplets, KML, and GeoRSS*. Expert's Voice in Web Development. Apress, New York, New York, 1 edition, November 2008.

[Wil12] Williams, Ed. *Aviation Formulary V1.46*, July 2012. ftp://ftp.bartol.udel.edu/anita/amir/My_thesis/Figures4Thesis/CRC_plots/Aviation.

[Yua10] Yuan, Jing; Zheng, Yu; Zhang, Chengyang; Xie, Wenlei; Xie, Xing; Sun, Guangzhong; Huang; Yan. T-drive: driving directions based on taxi trajectories. In *GIS '10 Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 99–108, New York, NY, USA, November 2010. Association for Computing Machinery.

[Zha07] Zhang, Jing Yuan; Shi, Hao. Geospatial visualization using google maps: A case study on conference presenters. In *IMSCCS*, pages 472–476, 2007.