Geospatial Search

BUILDING RICH FEATURES FOR VALIDATION, ANALYSIS AND SEARCH OF GEOSPATIAL DATA

LUCA GIOVAGNOLI
Geospatial Search

Building rich features for validation, analysis and search of geospatial data

LUCA GIOVAGNOLI

Master’s Thesis
Computer Science at the School of Computer Science and Communication (CSC)
Supervisor: Hedvig Kjellström (KTH), Elena Baralis (Polito)
Examiner: Danica Kragic
“The city’s central computer told you? R2D2, you know better than to trust a strange computer!”  

*C3PO*
Abstract

The main scope of this work is to develop enhanced search features over geospatial data in order to improve the user experience. The problem consists in finding a way to enable the entire geospatial dataset to be searchable. The goal is to add a new search parameter to the pre-existing full-text search query so that geospatial data is taken into consideration and better satisfaction of users needs is achieved. The report gives a clear theoretical overview of spatial data structures, while in the meantime relating their characteristics to various commercial applications, in the hope for the reader to use it as a reference and operate better informed choices with regards to geographical applications.

The method followed is principally the comparison between different solutions through the entire course of the work. Any single choice is weighted after thorough comparison of existing options, reading of papers, benchmarking or expert’s opinions.

Heuristics to assess quality of data and to achieve validation of data are created. The principal chosen solution is selected among a few commercial options and eventually enables the geographical data to be searchable. A Python web-service implementation is described in order for the search features to be accessible to the end users.

Experiments run on production code are presented to demonstrate the efficacy of the implemented heuristics. Benchmarking experiments show the validity of the solution for geo-searching.
Contents

1 Introduction
   1.1 Work outline ........................................... 1
   1.2 Hypothesis ............................................. 3
   1.3 Methodology ........................................... 3
   1.4 Report outline ....................................... 4

2 Related Work
   2.1 Overview of SDBMS implementation solutions .............. 7
   2.1.1 Spatial DataBase Management Systems or SDBMS ....... 7
   2.2 A medical application using GIS software ............... 9

3 Background
   3.1 R-trees ............................................... 11
   3.1.1 Building the tree ................................... 12
   3.1.2 Query processing ................................... 13
   3.2 Quad-Trees ............................................ 15
   3.2.1 Point Quad-tree .................................... 15
   3.2.2 Region Quad-tree .................................... 17
   3.3 Storing a collection of polygons using quadtrees ....... 20
   3.3.1 Decomposition criterias ............................... 22
   3.3.2 PM Quadtree: Point in Polygon search ............... 22

4 Naive approach method
   4.1 OpenGIS Consortium Standards ............................ 25
   4.1.1 Simple Features Specifications ....................... 25
   4.1.2 Well Known Text format and Well Known Binary Format 25
   4.2 PIP or point-in-polygon problem ......................... 28
   4.2.1 Ray-casting algorithm ................................ 30
   4.2.2 Considerations ....................................... 33
   4.3 Spatial coordinates and projection systems ............. 33
   4.3.1 SRID .............................................. 33
5 **Spatial data: commercial software**            35
  5.1 Solutions overview ........................................ 35
  5.2 Library solutions ........................................... 36
    5.2.1 Java Topology Suite .................................. 36
    5.2.2 Python Toblerity project and Pyproj .................. 36
  5.3 SQL DBMS: PostGres with PostGIS extension .............. 37
    5.3.1 Features .............................................. 38
  5.4 ElasticSearch .............................................. 38
    5.4.1 Architecture ......................................... 39
    5.4.2 ElasticSearch set up .................................. 42
    5.4.3 ElasticSearch strengths and defects ................. 46
    5.4.4 ElasticSearch used in real world applications .... 46

6 **Reverse Geocoder and experiments**             49
  6.1 Shapefiles ................................................ 49
    6.1.1 Process Shapefiles .................................... 50
    6.1.2 Dataset and conditions for the experiments ......... 52
  6.2 ElasticSearch benchmarks ............................... 52
    6.2.1 Elasticsearch precision parameter tuning .......... 53
    6.2.2 ElasticSearch bulk insertion test .................. 56
  6.3 PostGis benchmarks ....................................... 57
    6.3.1 PostGIS bulk ingestion ................................ 58

7 **Service Implementation**                      61
  7.1 Data Quality Assessment .................................. 61
    7.1.1 Google Maps API v3 .................................... 62
  7.2 Data validation ........................................... 65
    7.2.1 Heuristics ............................................ 66
  7.3 Set up a python web service .............................. 67
    7.3.1 Apache, Pyramid, Django, Ruby on Rails ............ 67
    7.3.2 Python package manager and virtual environment ... 68
    7.3.3 Pyramid set up ....................................... 69

8 **Results**                                      73
  8.1 Final benchmark comparisons ......................... 73
    8.1.1 Bulk ingestion results ............................... 73
    8.1.2 Search query results ................................ 73
  8.2 Geospatial software choice ............................. 75
  8.3 Service choices .......................................... 75
  8.4 Data processing achievements ........................... 75

9 **Conclusions**                                  77
  9.1 Methodology .............................................. 77
    9.1.1 Data pre-processing .................................. 77
    9.1.2 Data validation ...................................... 78
    9.1.3 Related work study ................................... 78
    9.1.4 Experiments ........................................... 78
    9.1.5 Web service implementation .......................... 78
  9.2 Future Work .............................................. 78
    9.2.1 Reverse Geocoder future complex street address version 79
List of Figures

1.1 Project outline — Map by Google. 2
2.1 Cluster and outliers analysis. 10
3.1 Example of R-tree topology. 12
3.2 Two steps query processing. 14
3.3 Point Quad-tree. 16
3.4 Point Region Quad-tree. 18
3.5 Smallest possible cell 19
3.6 PM1 Quadtree taken from [28]. 21
4.1 Example of LinearRing from document [14]. 27
4.2 Example of polygons with 1, 2 and 3 rings from document [14]. 27
4.3 Example of multipolygons from document [14]. 28
4.4 Polygon’s bounding box example. 29
4.5 Ray-casting algorithm. Odd number of intersection if the point is inside, even if the point is outside. 30
5.1 ElasticSearch general architecture from www.elasticsearch.org. 39
5.2 Cluster composed by a single node Gaia which contains five shards. 40
5.3 Cluster composed by two nodes Gaia and Joseph which contain five shards each (primary and replica). 41
5.4 Cluster composed by three nodes Gaia, Joseph and Ricochet with a clever distribution of shards. 41
6.1 Geometries associated to US zipcodes. 50
6.2 Features associated to US zipcodes geometries. 51
6.3 Elasticsearch ingestion times for the zipcodes database. 58
6.4 PostGis ingestion times for the zipcodes database. 60
7.1 Visualization of spatial data using Google Maps API v3. 63
7.2 Example of polygon with intersecting edges on map. 67
8.1 Comparison for ingestion times between Elasticsearch and PostGis 74
8.2 Comparison for search query between Elasticsearch and PostGis 76
List of Tables

2.1 Comparison Quadtrees and R-trees .......................... 8
3.1 Complexity for different types of quadtrees. .............. 19
5.1 Put Mapping Endpoint .......................................... 44
5.2 Insert data operation .............................................. 44
5.3 Elasticsearch query endpoints API. .......................... 45
6.1 Hardware specifications. ......................................... 52
6.2 Precision and Tree Level relation. ............................ 55
6.3 Elasticsearch Configuration ................................. 57
8.1 Search query times ................................................. 74
8.2 Data processing achievements. “Rough” is used because the standard deviation on the average is high. .......................... 76
Listings

4.1 Bounding box computation. . . . . . . . . . . . . . . . . . . . . 29
4.2 Point-in-bounding-box example . . . . . . . . . . . . . . . . . 29
4.3 Jordan Curve Theorem coded in C. . . . . . . . . . . . . . . . 30
4.4 Ray creation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.5 Line equation for a segment. . . . . . . . . . . . . . . . . . . . . 32
4.6 Solving a linear equation system. . . . . . . . . . . . . . . . . 32
4.7 Create table with geometry column. . . . . . . . . . . . . . . . 33
5.1 Mapping documents into ElasticSearch. . . . . . . . . . . . . 44
5.2 Example JSON document to be inserted into ElasticSearch. . . 45
5.3 Query to retrieve shapes that contain a certain point. . . . . . 45
6.1 Batch to convert the shapefile to GeoJSON in Python . . . . 50
6.2 Elasticsearch code for computing levels. . . . . . . . . . . . . . 53
6.3 Reverse engineer levels from precision. . . . . . . . . . . . . . . 54
6.4 Mapping documents into ElasticSearch. . . . . . . . . . . . . . 56
6.5 Document to insert into Elasticsearch. . . . . . . . . . . . . . . 56
7.1 Drawing polygons with Google Maps API v3. . . . . . . . . . 63
7.2 Pyramid configuration file . . . . . . . . . . . . . . . . . . . . . 69
7.3 Example view. . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
7.4 Example template. . . . . . . . . . . . . . . . . . . . . . . . . . . 70
Chapter 1

Introduction

The work described in this report addresses the problem of geospatial research, more specifically, building rich search features to make geospatial information searchable. All decision making processes are thoroughly explained in the hope that this work could serve as a reference when choosing and comparing geospatial options, and at the same time to give a clear overview on how the theoretical Computer Science details relate to the software applications in commerce. So the work tries to come in handy for understanding the connection between the theoretical geospatial data structure studied in college courses and the related software solutions in order for the reader to be able to make informed choices and use all their geospatial features at their maximum potential. Since it was noted lack of papers of this kind, the following one aims at being a first good attempt to give such an overview and such comparison information results.

1.1 Work outline

The project work described in the following document has been conducted during a six month internship employment in a software hi-tech company based in San Francisco.

The assignment consists in implementing new enhanced search features in order to improve customer’s search experience for food-delivering businesses. Let’s analyze a basic search experience for ordering food online. Most search engines allow the user to search for a business by inserting its name (or part of it). After landing to the business’ page, the user can then proceed with the order. This procedure has a couple of important flaws. For example, What happens if the business is unavailable? (the business may be too far from the delivery point or it might be too late in the night). A second issue might arise if the user does not have a clue about what to “google” for. He might not have a particular business in mind, because it does not matter how the delivery is performed but only that it is somehow. It is possible to build new search features to solve all these problems. The idea is to build a system able to present results to the query expressed in natural language: “return all businesses that can deliver to a certain given point.” A visual drawing of the idea for the final product is given in Figure 1.1.

Each business has geographical information associated to it (the service
The major amount of work interests the "Search" step. In order to be able to select all businesses whose service areas contain the given address, a linear scan of the dataset is an unfeasible option. The validity of the last assertion will be backed up by some experiments later in this work. The aim will be to reduce the time for the search to reasonable time complexity values. The immediate answer to the problem is to build an index over the data so to considerably reduce search times. This is the problem addressed by this work.
1.2 Hypothesis

The main hypothesis of this work is that it is possible to build geospatial rich search features over data presenting geospatial attributes so that the search process is scalable and well-performing.

The main objective is to demonstrate that it is possible to build such features and to implement them, making the right choices of professional tools and selecting the best configuration options. There are a few additional hypothesis made after the initial assignment of the task:

1. The data ingested by partner companies is completely unknown and might be different from expectations. A quality assessment process will come handy for validating this hypothesis.

2. The data ingested by partner companies might be broken. This will prove right and data validation will solve this problem.

3. A good rule in Computer Science affirms that re-inventing the wheel should be avoided. Implementing from scratch algorithms such as the ray-casting is useless and harmful.

4. There are many different possible options for implementing geospatial search features in 2014. Experiments to compare them will help finding the right one for this project. PostGIS is initially deemed to be the best due to some comparison results in literature and from expert’s opinions. This will be proved wrong, mainly due to the fact that PostGres on its own is not able to scale.

5. Elasticsearch ingestion time is deemed be inferior due to its schema-free architecture. This will prove wrong, being PostGIS unexpectedly faster during ingestion.

All the hypothesis here formulated will be kept in mind during the work and proved right or wrong. Additional in itinere assumptions will be introduced ahead in the document when appropriate.

1.3 Methodology

The thesis work is tackled as an engineering project, keeping in mind real-world needs for every choice made. A thesis of this kind focuses on building a new product or improving existing ones making them faster, fitter and better.

The experiments conducted will help making the right configuration choices among the various options and will help choosing the right tools. They will also be used to back-up statements and prove some of the hypothesis. The objective is for the final software to be scalable and well-performing for a Silicon Valley Big Data company. Therefore, as previously mentioned, the methodology used comprises executing related work studies to make informed choices, performing experiments to compare tools and try configurations, executing a background literature study to be able to understand the Computer Science concepts at the base of the implementations, so to be able to use them at their best.

In details the following is a summary of the choice of methodology used. Methodology will be more thoroughly described in the “methods” chapters 4, 5, 6, 7.
Data pre-processing

Since the data ingested by partner companies is initially unknown in our own databases, the hypothesis that it might be unfit or different from what expected is very realistic. A quality assessment will be performed and a visualization tool will be built to verify this. The data will be proved very different from what expected indeed. This method comprises Data qualitative audit and Data quantitative audit steps presented in Chapter 7.

Data validation

Regardless the nature of the data, another hypothesis is that a consistent percentage of the data might be invalid. This will be proved right after the qualitative audit. The problem will be solved by modifying the validation code introducing new heuristics to increase validation percentage from about 70% to about 97%. The new validation heuristics will be put at Data Ingestion time.

Experiments

The experiments conducted will be very useful for proving different assumptions. For example, timing results for ingestion times will show that Elasticsearch is unfortunately slower than PostGIS during ingestion. Luckily ingestion is a process performed once and for all, so it will not impact our final decision of Elasticsearch as a solution. Other timing experiments will be used for proving that Elasticsearch is actually faster than PostGIS for querying operations and that it scales better. They all will be also used for tuning different configurations and finding the right one.

Related work study

This method will be heavily used in order to compare solution without having to try them out ourselves. All the decisions taken will be preceded by careful study of related works where similar choices were made. This process is very well-know and it is sometimes taken to extremes by companies following the Agile methodology.

To make an example, the method used for the decision-making during the web-service implementation will be a thorough comparison of existing solutions and softwares made by reading related works or expert’s opinions from papers, blogs or books. This method will prove very efficient mainly because it avoids repeating work already done by others (again for the golden rule in Computer Science mentioned above).

1.4 Report outline

The report and work outline are presented in this introductive Chapter. A literature study on related work is in Chapter 2 where papers are presented to show an overview of DBMS systems including geospatial features and a use case of GIS softwares used in medical applications. Geospatial algorithms are
described in Chapter 3 to provide background for this report and the historical development of geospatial data structures. Commercial software solutions and implementation details will be introduced in Chapter 5. Experiments about benchmarking different solutions will be presented in Chapter 6 while developing a Reverse Geocoding application. Finally, in Chapter 7 the engineering process of a Python web service will be described, along with all the planning decisions that brought to different design choices. In conclusion, results will be summarized and some interesting ideas for future work will be briefly treated in the last Chapter 9.
Chapter 2

Related Work

This chapter presents a few papers treating related works. The first paper taken into consideration [15] regards an overview, comparison and benchmarking of different SQL DBMS solutions with geospatial features. This dissertation tackles a different approach. While paper [15] focuses on SQL solution, Chapter 5 and Chapter 6 compare an SQL vs. noSQL (key-store value) solution. The second paper [25] offers an example of a real world medical applications where GIS software is used. The GIS software QuantumGIS is used in Chapter 6 in order to help visualizing and analysing data from shapefiles.

2.1 Overview of SDBMS implementation solutions

It is preposterously hard to find technical information about implementation details of current DBMS solutions, either because they are proprietary, or simply due to lack of documentation. Moreover, it is even harder to find this kind of details with regards to spatial modules. In fact, spatial advanced features have only been quite a recent addition to database systems, also thanks to the major interest developed around localization in recent times. Mainly due to these reasons, paper [15] is chosen to be cited in the “Related Work” section. Its aim is to discuss current research efforts towards better support for handling 3D spatial data, given the shortcomings in nowadays technology. Although the main topic is hardly inherent to the present project, the paper offers a satisfying overview of the state-of-the-art in the geospatial field.

2.1.1 Spatial DataBase Management Systems or SDBMS

The SQL database systems presented in the work are: Oracle, PostGis and SQL Server. Since SQL Server introduce spatial features only recently (SQL Server 2008) and their nature is very basic they were not treated in the paper. A similar reason may be adduced with regards to MySQL, whose spatial indexing at the version 5.6 was still not able to compete. Both Oracle and PostGres added modules to their existing DBMS: Oracle’s one was called Oracle Spatial and introduced with Oracle 8i, while PostGres enhanced PostGres providing spatial functionalities in the early years of this millennium.
The implementation solutions offered by SDBMS have historically happened to be one of two main categories: quadtrees as developed by Samet or the Rtrees by Guttman (reference to papers before in this Chapter). As previously seen, there exists many many variations of both quad and R-trees: point quadtrees, region quadtrees, k-d trees, R-trees, R+trees, T+trees (and so on) and as such, different types of implementations have been historically used by SDBMS. Often the nowadays vendor implementation are the final result of a long development of a certain historical product. Both Oracle Spatial and PostGis use R-trees as main implementation solution.

“The R-tree structure was developed to overcome shortcomings of existing indexing structures at the time (Guttmann, 1984). Cell structures, for instance, are not dynamic, as the cell size has to be decided in advance. K-d trees, on the other hand, are designed particularly for point data (Bentley, 1975) and use paged memory.” [15]

Oracle offers the opportunity to use a quadtree index. This is very well documented in their Oracle Spatial 9i Reference Manual. It is considerably worth noticing that the quadtree option was removed by the Manual in the last versions [22].

**Table 2.1: Table taken from [15]**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Quadtree(tiling level 8)</th>
<th>R-tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>10.507 s</td>
<td>51.015 s</td>
</tr>
<tr>
<td>Update</td>
<td>675 s</td>
<td>267 s</td>
</tr>
<tr>
<td>Storage</td>
<td>22.725 MB</td>
<td>2.060 MB</td>
</tr>
</tbody>
</table>

Comparison between R-trees and Quadtrees in SDBMS

The paper offers a nice quick performance and storage comparison chart between Quadtree and R-tree after applying the basics operations of Insert and Update. The dataset used: 50.000.000 LIDAR points. Quadtree tiling level: 8 Computer configuration: Intel Pentium 4 CPU 3.2 GHz, 2GB DDR2 RAM, 7.200RPM 300 SATA hard drive on Oracle 11g 32 release 11.1.0.6.

From the table we can observe results that agree with the general theory: R-tree are slower to create (about 5 times), while they are faster in update and they are enormously more convenient in terms of storage space (around 10 times less storage required). Obviously, results depend on geometry inserted and queries executed (as seen previously in this same Chapter there are many types of different spatial queries). From theoretical results according to [15] R-tree perform way better on insertion of large polygons.

Finally, it is worth noticing that PostGis is an open-source tool, (contrarily to Oracle Spatial) which might affect the decision about which one to use.
2.2 A medical application using GIS software

Following is an example of how GIS softwares can be also used for applications conceptually very far from the fields that they were engineered for. It seems amazing how unexpected good uses these complex softwares can be put for. The paper was found during a literature research about how Data Mining could be applied to Geospatial Data. The search did not yield many results. The state of the art of data mining for Geo-data is very young, the information retrieval for this kind of data is still mainly performed manually because it yields the best results. The following paper about bones micro-structure is only presented as a curiosity to show how GIS methods can be useful and applied to a diverse range of fields.

On September 26th, 2012 it was published a paper on the use of a commercial GIS software called ArcGIS platform (by ESRI) for a medical application. Although it was not necessarily the first time that a software of this kind found application in the medical field, was it the first indeed that a GIS software was used to map bone micro-structure. Paper [25] presents the work’s results. The innovative idea was to analyze the distribution of micro-structure in the bones while considering the impact that load history had at the macroscopic level. In other (more simple) words, the software was successfully used to extract patterns describing the evolution and adaptation of a metatarsal bone (technical word for the foot’s bones) taking into account the load history (how the foot was stressed). From a computer science point of view, the clustering operator was mainly used (k-nearest neighbours algorithm) in order to identify and classify patterns. For example in Figure 2.1 we can see cluster and outliers analysis on bones osteons\(^1\). Black dots in the picture are outliers while the bulls-eyes represent osteons with high morphotype score. The morphotype score was defined by Martin in 1996 [25] and it is used to assign a number from zero to five to osteons who present fiber formed under compression (close to five) or under tension (close to zero).

From these patterns it was possible to connect the bone’s microstructure situation to possible skeletal diseases like bone fragility or osteoporosis\(^2\). Other possible applications can be found in the forensics field. Sometimes, when human remains are discovered after a long time, only bones are left intact. Following with the results of this study, it will be possible for researcher to identify details such as sex, age, body size, looking at the patterns in the bone structure, using our classification model to relate them to information that might be of use for the law enforcement.

\(^1\)The osteon is the fundamental unit of compact bone, see \url{https://en.wikipedia.org/wiki/Osteon}

\(^2\)Sam Stout from \url{http://researchnews.osu.edu/archive/osteons.htm}
Figure 2.1: Cluster and outliers analysis.
Chapter 3

Background

Before diving into the main core of the work, it will follow a generic introduction on the most common spatial data structures and algorithms, among which: R-trees and Quad-trees (in many of their flavours). Benchmarks will follow in the experiments Chapter 6.

3.1 R-trees

R-trees are a data structure used for creating indexes over spatial data proposed by Guttman in 1984. Bidimensional data only will be considered here. The R-tree is able to store geometries of arbitrary shape (i.e. points, polygons, multipolygons) and execute queries of different nature over the indexed geometries. A complete specification for the OpenGIS geometry model can be found at [6].

Like the more widely known B-tree, the R-tree is a balanced tree, meaning that:

1. all leaves of the tree are at the same height.
2. a NodeSplit algorithm is necessary when the children in any node reach the maximum allowed.

The general idea to handle geometries in a performant manner is to represent them with their MBR. The MBR is the Minimum Bounding Rectangle of a geometry. A MBR is easily identified by two points, any two opposite corners of the rectangle. To make an example from a real-life application, in the C++ implementation of the R-tree living inside the library libspatialindex [11] the MBR is identified by the top left corner and the bottom right corner.

The nodes of the R-tree can belong to any of two categories: internal node, or leaf node. An internal node contains a list of children and an MBR. (the MBR contains all the geometries of the children). A leaf node will contain only the MBR of the geometric shape of that leaf, plus a pointer to the real database entry of the shape containing information about the exact shape.

In Figure 3.1 we can see an example of R-tree structure. For simplicity only rectangular shapes are indexed (the ones in red colour). The intermediate nodes are indicated in blue colour, the root entries are black. It is clearly
noticeable how the MBR of internal tree nodes are overlapping (differently from other r-tree versions). More details about R-trees can be found at [19].

3.1.1 Building the tree

In order to build the tree, algorithms for insertion, deletion, node splitting must be used. They will be quickly outlined in order to have an idea of the algorithmic complexity implicitly carried by using a data structure of this type.

Insertion algorithm

The insertion algorithm general idea is to execute a search query over the index in order to get a list of MBR that overlap with the shape to insert. When all nodes whose MBR overlap the shape to be indexed have been selected, the next step is to modify the MBR to contain the new shape, and save the oid (object identifier) to retrieve the exact data from the database.
Node splitting algorithm

It may happen that the maximum of children per node is reached after an insertion operation. In such cases, a node splitting algorithm is needed. There are historically three algorithms for node splitting in R-trees that were proposed by Guttman.

1. **Linear Splitting:** linear splitting is the fastest algorithm among the three but it is also the one that achieves the worst splitting final combinations, bringing to an higher time required for searching. Technically, two object as far apart as possible are chosen as first elements of the two new groups. Afterwards, all other objects are assigned to either of the groups, choosing the one where the least MBR enlargement is needed.

2. **Quadratic Splitting:** Quadratic Splitting is a compromise between linear and exponential splitting. In practice it searches for the pair of nodes that is the worst combination to be in the same group and puts it as first entries for the two groups. Afterwards the algorithm searches for the object that is the best fit for being inserted in either of the groups. The operation is repeated for each remaining object. Complexity is $O(n^2)$. In practical applications, Quadratic Splitting is usually used, since it offers the best trade-offs for index-creation-time and index-search.

3. **Exponential Splitting:** Exponential Splitting consists in trying all possible combinations for creating the two groups. It is the most expensive Node Splitting algorithm but it is also the one that brings the best benefits during search.

3.1.2 Query processing

In order to talk about query processing, we need first to write down a list of the possible existing queries that can be run on the index.

- **Topological operators:**
  - Disjoint
  - meet
  - overlap
  - covers
  - contains
  - inside
  - equal

- **Directional operators:**
  - Example query: “Find all objects that lie north of a given input object”.

- **Distance operators:**
  - Range query: “Find all objects that lie a certain distance from a given object”.
  - k-nearest-neighbours query: “Find the k nearest objects that lie a certain distance from a given object”.
During the project work described in this thesis, only topological operators have been used (in particular, the “overlap” operator). We will therefore generally introduce and outline only the former group.

Query processing in R-trees happens in two steps, called “filtering step” and “refinement step”. The filtering steps consists in defining a set of candidates while the refinement step is needed to actually identify the results among the candidate set.

In Figure 3.2 is a schema of the query processing flow. Notice that we get “hits” after the filtering step only for some of the query types, i.e. directional operators, where determining that the MBR of an object is north of another MBR is enough to get the answer to the query.
Filtering Step
At first we have the filtering step where the intersection between the given shape and the MBRs is checked. It is created a candidate set of all possible MBRs intersecting the MBR of the input shape. We remind that the MBR is an oversized approximation of the original indexed shape so all the results in the candidate set are potentially results for the query but not necessarily.

Refinement Step
The refinement step is necessary to check if the exact shape in input intersects the exact shapes in the candidate set. It is to notice that the intersection between exact shapes is computationally way more expensive than checking intersection between MBRs. This is a big advantage of using MBRs for the internal nodes and checking the exact shapes only at the end, after intensive filtering has been applied.

Searching
Search operations though an R-tree are basically performed identically, regardless the R-tree version implemented. Different R-tree versions (such as dynamic R+trees or R*trees) mainly differ for the heuristics chosen for the node splitting procedures.

3.2 Quad-Trees
Quad-trees are a hierarchical data structure for indexing bi-dimensional data. Most of the theory about Quad-trees was developed during the 70s by the work of Klinger, Finkel, Bentley and Hunter. In particular Finkel and Bentley were responsible for the creation of the Point Quadtree. To Bentley alone is also attributed the creation of the k-d tree. A detailed description of spatial data structure (and quad-trees especially) can be found in the classic textbooks of Hanan Samet, University of Maryland[26]. The quadtree can be considered the bi-dimensional case of a binary tree and this concept can even be generalized to the case of more than two dimensions. So we have binary trees in one dimension, quadtrees in two, oct-trees in three dimensions (used in graphical applications).

3.2.1 Point Quad-tree
In general, quad-trees achieve the same spatial purpose as the R-trees described in the last section, but they are based on a slightly different idea. Here the point quad-tree is presented as a first step to get to the region quad-tree, matter of interest for this project. While R-trees are a general case of a B-tree in more dimensions, Point Quad-trees are a general case of a Binary Search Tree in two dimensions. Similarly to a binary tree, comparisons are applied at every level of the tree in order to decide which quadrant to continue on (both for insertion, search and other basic operations). For a point quadtree the quadrants are named with cardinal directions’ names NorthEast (NE),
NorthWest (NW), SouthEast (SE) and SouthWest (SW). A point quad-tree is visible in Figure 3.3

Each node in the point quad-tree stores:

- $d$ coordinates ($d$ number of dimensions). For the quad-tree it would be $x$ and $y$.
- $2^d$ pointers to the quadrants. For the quad-tree there are 4 quadrants.
- Pointer to the data.

**Insertion, deletion and search algorithms: general idea**

The general idea for the insertion phase is to choose a point as root of the tree and insert all the others one after the other. At every level, we compare the coordinates in order to decide which quadrant to descend on, until we reach a leaf node where to insert the new one. In this kind of tree a node represent a region. The mechanism is exactly the same as in binary search trees with the slight difference that the number of comparisons per level depends on the number of dimensions $d$.

The search algorithm is logically the same as insertion but we keep iterating until we find the solution or a leaf node (no solution).

**Deletion**

Deletion is by far more the most complex among the operations and will be skipped because not inherent to this work. From an historical point of view, Klinger and Bentley proposed that all the nodes under the deleted one should be reinserted but later some more efficient techniques were studied by Samet [26].
Complexity analysis results for Point Quad-trees

Insertion for a point quad-tree takes $2 \cdot n \cdot \log_4 n$ in the average case (we insert $n$ nodes in a tree with a branching factor of four). The generalized formula is $d \cdot n \cdot \log_{2^d} n$ where $d$ is the number of dimensions. As well known from basic results about the analogue binary tree studied in basic computer science courses, a tree of this type can become very inefficient in the worst case scenario. In fact, the tree might be unbalanced yielding to insertion times of $O(n^2)$. In case the entire dataset is known a priori though, it is possible to do better and performance of $O(n \cdot \log n)$ in the worst case can be reached. The procedure involves sorting the nodes by one coordinate (say x) as the primary key and the other (say y) as the secondary key. At this point the median point is chosen for insertion. This will guarantee that not more than half the nodes will end up in one of the four quadrants. The procedure is repeated for each quadrant: first sort all the points, then store the median. Building a sorted structure of this kind to find the median takes $O(n)$ time at each level, and must be repeated for $O(\log n)$ levels (where $h = \log n$ is height of the tree).

Search in a point quadtree is bounded by $O(\log n)$, as well as point insertion when the tree has been already built. To be more precise for the quadtree we have a formula of $O(d \cdot h)$ where $h = \log_{2^d} n$ is the height of the tree.

3.2.2 Region Quad-tree

Region quad-tree main difference with respect to the point quad-tree is the subdivision of space into cells regardless the dataset. Let here be defined the length of a cell as:

Length of a cell: it is the edge’s length of a cell in the quad-tree.

Space is successively partitioned in square cells, whose “length” is always half of the parent cell. The first cell dimensions are identified so that all the data can be contained inside the root. Therefore, the root cell length depends on the distance between the two farthest points in the dataset. Space is then partitioned into four cells of equal length: NorthEast (NE), NorthWest (NW), SouthEast (SE) and SouthWest (SW). Each one of these cells is partitioned again into four and the procedure goes on until every single point in the dataset is isolated into only one cell. Basic operations are straightforward. For search, the procedure consists in traversing the tree until we find an entry or a leaf. Tree traversal is performed descending through the right quadrant at each level. Nodes are situated in the trees. For insertion, we descend through the path that should contain the new node until the leaf. At this point we start splitting the leaf into quadrants until the old leaf node and the new one are divided, each in a different quadrant. For deletion we need to execute a search for the node and delete it. At this point, in case the node was the only child of its parent, we need to traverse the tree upwards until we find a node with two children and delete the path constituted by nodes with a single child.

An example of a PR (Point Region) Quad-tree is visible in Figure 3.4.
Region quadtree height: worst case scenario

It is worth noticing that the closest the minimum distance between any two cells in the tree, the highest the number of subdivisions needed to isolate them. The worst case scenario is achievable with only three points unlucky positioned into the space: two of them very close to each other and the third one very far from these two. Since the initial cell needs to contain all the points, its length is equal to the maximum distance between any two points. Moreover, a very high number of subdivisions (and thus, of levels) is needed in order to isolate the other two points.

Let’s define as $s$ the smallest distance between any two points and as $D$ the length of the root cell. Let’s remember that the root cell is chosen such that all the dataset points are contained in it. Hence, the maximum distance among any two points will be equal to $\sqrt{2} \cdot D$. We will use $d$ for the number of dimensions as before. The length of the smallest cell able to divide two points at distance $s$ is $\frac{s}{\sqrt{2}}$ (case when the points are on two opposite corners of the cell). Let’s call $h$ the number of subdivisions or levels (or height) of the tree. We must then have that

$$\frac{D}{s^{\frac{d}{2}}} < \frac{s}{\sqrt{2}}$$

In words, we must make sure that the length of a cell must be less or equal than the maximum allowed value. If we solve the inequation we get:

$$h > \log \frac{D \cdot \sqrt{2}}{s}$$

This means that in the worst case scenario, the height of the quadtree is bounded by $O(\log \frac{D}{s})$, a measure of the non-uniformity of the distribution of the dataset points. Hence, we obtain the result stated before that the height
Figure 3.5: Smallest possible cell

<table>
<thead>
<tr>
<th>Tree type</th>
<th>Search</th>
<th>Insertion</th>
<th>Deletion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point Quadtree</td>
<td>$O(\log_4 n)$</td>
<td>$O(\log_4 n)$</td>
<td>$O(\log_4 n)$</td>
</tr>
<tr>
<td>PR Quadtree</td>
<td>$O(\log \frac{D}{s})$</td>
<td>$O(\log \frac{D}{s})$</td>
<td>$O(\log \frac{D}{s})$</td>
</tr>
</tbody>
</table>

Table 3.1: Complexity for different types of quadtrees.

(and all operations depending on it) depend on the maximum distance between any two nodes, divided by the smallest. For the region quadtree, the performance is affected by the distribution of the data points. Alternatives where studied and are nowadays used, such as the **Compressed region quadtree** which attains benefits from both region quadtrees and point quadtrees. The appellative *Compressed* comes from the fact that the tree is able to save a compressed representation of the structure with paths compressed whenever possible. This brings to a smaller height of the tree at the cost of a more complicated representation of it.

**Complexity analysis results for Point Region Quadtrees**

Complexity analysis for the PR Quadtree is quite straightforward. Search, insertion and deletion for the PR Quadtree take each $O(h)$ where $h = \log \frac{D}{s}$. Insertion can be done in bulk if the dataset is known a priori or dynamically. Either case the complexity is the same, insertion takes $n \cdot O(\log \frac{D}{s})$. 
3.3 Storing a collection of polygons using quadtrees

All the quadtrees briefly introduced in Chapter 1 are meant to be used with point data. Unfortunately, there is great scarcity of information about how to use quadtrees with polygonal data (probably due to the high-specificity of the task). Due to this lack of information, it has been considered best to present here one of the original numerous papers by Hanan Samet on the matter. Hanan Samet is somehow considerer the “father” of quadtrees. As from the online encyclopedia\(^1\):

“Samet is a pioneer in research on quadtrees and other multidimensional spatial data structures for sorting spatial information.”

Paper [28] presents an adapted version of a quadtree that is well fit to store polygonal maps.

The paper takes a progressive (“evolutionary” to cite it) approach, introducing the final data structure in three main steps. Initially a data structure able to solve the problem is introduced with relaxed constraints. At each step, the data structure is improved and the constraints are made stricter. The three data structures introduced are indicated with \(PM_1, PM_2, PM_3\), where \(PM\) stands for Polygonal Map.

The paper tackles the problem with three main use cases in mind:

1. **Point location problem**: the objective is to locate which region contains a given pair of coordinates. In the quadtree data structures there are no regions represented (as instead it happens in R-trees), hence the problem translates into finding a segment that is the border of the region containing the point. We will see that the goal will be to determine on which side of the segment the pair of coordinates lie.

2. **Dynamic line insertion problem**: it consists in the problem of adding new segments to an already existing structure, an important topic in computer graphics.

3. **Map overlay problem**: this problem can be considered a generalization of the previous. Instead of inserting a single line, it considers a collection of lines.

Out of the three reasons, the point-in-polygon determination is exactly the problem we are looking forward to solving in this work.

The paper posed as main goals:

1. The data structures should be able to store the polygonal data without information loss of any kind (no loss of accuracy due to digitization).

2. The quadtree should not be suffering from changes due to the position of the map. In other words, operations such as shift or rotation do not drastically degrade space complexity.

We are mainly interested to the first of the two problems. How to store the polygonal map without loss of information and how to keep low times for search operations. In order to understand the subsequent discussion, we need to introduce the definition of *q-edge*. Quoting the paper:

\(^1\)https://en.wikipedia.org/wiki/Hanan_Samet
Figure 3.6: PM1 Quadtree taken from [28]
“We use the term q-edge (denoting a quadtree-decomposition edge) to refer to segments that are formed by clipping an edge of the polygonal map against the border of a region represented by a quadtree node.”

For example, EF and FG in Figure 3.6 are q-edges. With this in mind, we can introduce the three criteria used by Samet in order to define the PM Quadtree. They will be called decomposition criteria because they will be dictating the rules for how to decompose the quadtree quadrants.

### 3.3.1 Decomposition criterias

- **C1**: At most one vertex can lie in a region represented by a quadtree leaf.
- **C2**: At most one q-edge can lie in a region represented by a quadtree leaf.

Criteria C1 means that the decomposition will go on until there is at most one vertex per quadrant. According to C2, similar rules apply for q-edges. Decomposition of the tree in quadrants will go on until there is at most one q-edge per quadrant. Unfortunately, this means that decomposition could go on virtually forever, due to the edge case when we have a vertex with two q-edges connected to it. If the vertex will not fall exactly on the border of a quadrant we may need to keep decomposing the three beyond a reasonable level. Hence, it is necessary to introduce the new criterias **C2′** and **C3** to substitute to **C2**.

Quoting the paper:

- **C2′**: If a region contains a vertex, then it can contain no q-edge that does not include that vertex.
- **C3**: If a region contains no vertices, then it can contain at most one q-edge.

The new criterias allow the presence of a vertex and multiple q-edges connected to this vertex in a single leaf node. If there is no vertex, only one q-edge is allowed in a leaf node. We have now the three criterias **C1**, **C2′**, **C3** that we can use to build the tree. This tree will be called **PM₁** quadtree and it is shown in Figure 3.6. At this point, the paper presents a complexity analysis for the **PM₁** quadtree, proposing then two other alternatives, the **PM₂** and **PM₃** quadtrees. The two latter try to optimize space, so they are not very inherent for us and hence will be skipped. We will instead extensively treat how to search for all polygonal maps containing a given point in **PM₁** quadtree data structure.

### 3.3.2 PM Quadtree: Point in Polygon search

This is the main point of the entire study. Our goal is in fact to find out how point in polygon search is achieved with a quadtree data structure. First of all we need to perform a normal search for the point so to obtain the leaf where it is located. We can fall in one of three possible cases, illustrated in Figure 3.6 as points X, Y and Z:

1. **Point X**: case when the leaf contains only a q-edge. At this point, the next step for the algorithm will be to identify which side of the q-edge
the point is. Since each q-edge is stored together with the information about the regions that it is dividing, this is a trivial task.

2. **Point Y:** _case when the leaf contains a vertex, and thus possibly multiple q-edges._ We can easily simplify this case to the one of determining which are the neighbours of the q-edge passing through Y and C (C is the vertex in the example in Figure 3.6). This task is quite easy thank to the data structure used to store the q-edges. It can vary but usually it is used some kind of dictionary from which we can easily obtain the sequence of sorted q-edges (for example sorted by the angle that they form with the X Cartesian axis).

3. **Point Z:** _case when the leaf does not contain vertices nor q-edges._ This case can be easily conducted back to the formers thanks to how the $PM_1$ is built (see paper [27]). The criterias for building the tree are such that one of the brothers of our leaf node must contain at least a q-edge. Hence the algorithm just needs to iterate through the siblings of the leaf node. If they are empty, it keeps iterating. When it finds a non-empty node, it will just repeat steps 1 or 2, depending on which is the case (q-edge or vertix). Only modification will be to introduce a point $Z'$ inside the node that is infinitesimally close to the leaf node and execute with $Z'$ as input.
Chapter 4

Naive approach method

This chapter is an introduction to the first approach to tackle the problem and a sort of engineering background for the successive chapters. First of all, in order to establish a common ground for the terms used from this point on, the OpenGIS Standards for addressing geometries are described in the first section 4.1. Afterwards, a naive approach to the problem is presented, even though completely unfeasible in real life applications. Finally, in the last section, spatial coordinate projection conventions (SRIDs) are described in order to get a better hold of the following Chapter 5, which treats commercial solutions using the same SRIDs.

4.1 OpenGIS Consortium Standards

The Open Geospatial Consortium (also OGC) is an international standards organization that was created in 1994. The standards defined are open and focused on geospatial content and GIS data processing. The consortium is composed by more than 500 companies, universities and agencies.

For a list of the OGC standards, see http://www.opengeospatial.org/standards

4.1.1 Simple Features Specifications

In this section we present one of the numerous OGS specification documents, the “Simple Feature Specification for SQL”. The document is of interest for the present work mainly because it is necessary to be very confident with the GIS specification for data representation when handling geospatial data.

4.1.2 Well Known Text format and Well Known Binary Format

WKT and WKB are formats widely used in the GIS field to describe geometry objects. We can find them used in the shapefile format as well as in modern database systems like MySQL or PostGIS. The standards were defined by the Open Geospatial Consortium (OGC) in the ISO/IEC 13249:2011 “Information
There are 18 distinct representable geometry objects: Geometry, Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon, Triangle, CircularString, Curve, MultiCurve, CompoundCurve, CurvePolygon, Surface, MultiSurface, PolyhedralSurface, TIN, GeometryCollection.

A few examples among those of our interest are listed now. There will be shown pictures and WKT representation for polygons. WKB representations are omitted given that there would make little sense since they are not human-readable.

**POINT**

A point is the simplest geometry to represent. It is composed by only two coordinates. Here is its WKT representation:

\[
\text{POINT}(x_1 \ y_1)
\]

**LINEARRING**

A LinearRing is a closed LineString.

\[
\text{LINEARSTRING}(x_1 \ y_1, x_2 \ y_2, x_3 \ y_3, x_4 \ y_4, x_1 \ y_1)
\]

It is worth noticing that the LinearRing is a sequence of coordinates (hence a LineString) that represent a closed figure (it starts with the same point as it ends). An example of a LinearRing in Figure 4.1.

**POLYGON**

A Polygon is composed by an exterior ring and an internal ring. The rings are of type LinearRing (see [14] for details on LinearRings). It is hence possible to define polygons with holes inside. The specification requires a polygon to be topologically closed. A polygon representation is a list of LinearRings. The first one is the exterior ring. Each following one represents interior holes. An example of valid polygons in Figure 4.2.

The Well-Known Text representation of a Polygon is:

\[
\text{POLYGON}((x_1 \ y_1, x_2 \ y_2, x_3 \ y_3, x_4 \ y_4, x_1 \ y_1), (x_6 \ y_6, x_7 \ y_7, x_8 \ y_8, x_6 \ y_6))
\]

where \(x_i\) and \(y_i\) with \(i=1,2,...\) are coordinates. Notice the presence of a first LinearRing (geometrically that would be the external border of the polygon) and a second LinearRing which represents a hole in the polygon. Notice also the fact that the LineStrings are closed (the first one starts with \(x_1 \ y_1\) and it ends with the same).

**MULTIPOLYGON**

A multipolygon is a collection of polygons which cannot intersect among each other. An example of valid multipolygons in Figure 4.2.

The Well-Known Text representation of a Polygon is:
Figure 4.1: Example of LinearRing from document [14].

Figure 4.2: Example of polygons with 1, 2 and 3 rings from document [14].
MULTIPOLYGON(((x1 y1, x2 y2, x1 y1)), ((x4 y4, x5 y5, x4 y4)))

where $x_i$ and $y_i$ with $i=1,2,..$ are coordinates. Notice the number of parenthesis. In this case the multipolygon is a list of polygons which have not any interior ring. Each polygon is a list of LinearRings (hence the double parenthesis for a polygon). The Multipolygon itself is a list of polygons (hence the third parenthesis).

### 4.2 PIP or point-in-polygon problem

The entire work can be seen as centred to the point-in-polygon problem, a famous computational-geometry problem. Algorithms to solve this problem were in use already in 1974 [30]. There exist a couple of solutions to the algorithm. The ray-casting one will be presented together with an implementation developed in C.

First of all let’s introduce the concept of bounding box for a given polygon. The bounding box is the minimum rectangle able to contain the polygon(see Figure 4.4 for an example). A simple way to find the bounding box is to compute the minimum and maximum x coordinates for the polygon and the minimum and maximum y coordinates. These four values define the rectangle. Sample C code for bounding box at 4.1.
BOUNDING_BOX compute_bounding_box(POINT* polygon, int N){
    int i;
    BOUNDING_BOX bb;
    bb.min_x=DBL_MAX; bb.max_x=DBL_MIN;
    bb.min_y=DBL_MAX; bb.max_y=DBL_MIN;
    for(i=0; i<N; i++){
        if (polygon[i].x < bb.min_x) bb.min_x = polygon[i].x;
        if (polygon[i].x > bb.max_x) bb.max_x = polygon[i].x;
        if (polygon[i].y < bb.min_y) bb.min_y = polygon[i].y;
        if (polygon[i].y > bb.max_y) bb.max_y = polygon[i].y;
    }
    return bb;
}

Listing 4.1: Bounding box computation.

So the first check is trivial. If the test point is not in the bounding box, it is impossible that it may intersect the polygon. See listing 4.2.

int is_point_in_bounding_box(BOUNDING_BOX bb, POINT test_point){
    if(test_point.x < bb.min_x || test_point.x > bb.max_x) return FALSE;
In case the point is inside the bounding box though, there might be the chance that it is inside the polygon as well. Hence a more thorough check must be applied. Fortunately, the bounding box check prunes away a lot of work.

### 4.2.1 Ray-casting algorithm

The basic idea is that we can cast a ray starting from the test point towards infinity (a fancy way to say “outside the bounding box”). It is important that the ray start from the point towards another point outside the bounding box. At this point we just need to count how many times the ray intersect any edge of the polygon. If the number is odd, the point is inside, otherwise it is outside. This theorem is called the Jordan Curve theorem and the formal demonstration can be found at [21]. A visualization of the ray-casting step is in Figure 4.5, while listing 4.3 is a C implementation.

```c
int is_point_in_polygon(POINT* polygon, int N, POINT test_point){
    int i, intersection_cnt=0;
    if(test_point.y < bb.min_y || test_point.y > bb.max_y) return FALSE;
    return TRUE;
}
```

Listing 4.2: Point-in-bounding-box example
BOUNDING_BOX bb = compute_bounding_box(polygon, N);

/*checking at first if the point is in the bounding box saves a lot of time*/
if (is_point_in_bounding_box(bb, test_point)==FALSE) return FALSE;

/*create the ray from test_point to a point outside the bounding box*/
SEGMENT ray = create_ray(test_point, bb);
for (i=0; i<N; i++)
    SEGMENT seg = {polygon[i], polygon[(i+1)%N]};
    if (segments_intersect(seg, ray)==TRUE )
        intersection_cnt++;
}
if (intersection_cnt%2 == 0) return FALSE;
return TRUE;

Listing 4.3: Jordan Curve Theorem coded in C.

The only problem left to solve is how to determine the intersection between segments. Then we can use the procedure to check intersection between the ray and each of the polygon edges, one after the other. A ray is just a long segment created starting from the test point to a point outside the bounding box. It can be created as in 4.4 where a padding value \( e \) is used to be sure that one end is outside the box.

SEGMENT create_ray(POINT test_point, BOUNDING_BOX bb){
    /*the ray must start OUTSIDE the bounding box*/
    double e = (bb.max_x - bb.min_x);
    SEGMENT s = {{test_point.x+e, bb.max_y+e},
                 {test_point.x, test_point.y}};
    return s;
}

Listing 4.4: Ray creation.

In order to get the intersection point between two segments we can use some basic results from elementary algebra. First of all we need to get the equations for the lines passing through the segments. Given the standard form for the linear equation:

\[ a \cdot x + b \cdot y = c \]
we can find $a$, $b$ and $c$ using any two points passing through the line (or segment) solving the following linear system:

$$\begin{aligned}
a &= y_2 - y_1 \\
b &= x_1 - x_2 \\
c &= a \cdot x_1 + b \cdot y_1
\end{aligned}$$

where $(x_1, y_1)$ and $(x_2, y_2)$ are the coordinates of the two points $p_1$ and $p_2$. We can code this like in 4.5.

```c
LINE segment_to_linear_equation(SEGMENT s){

    LINE line;
    line.a = s.p2.y - s.p1.y;
    line.b = s.p1.x - s.p2.x;
    line.c = (line.a) * s.p1.x + (line.b) * s.p1.y;
    return line;
}
```

Listing 4.5: Line equation for a segment.

Given the two linear equations, we can now solve the following linear equation system:

$$\begin{aligned}
a_1 \cdot x + b_1 \cdot y &= c_1 \\
a_2 \cdot x + b_2 \cdot y &= c_2
\end{aligned}$$

The solution $(x, y)$ is the intersection point. We can code this like in 4.6.

```c
int solve_system(LINE l1, LINE l2, POINT* solution){

    double det = (l1.a*l2.b) - (l2.a*l1.b);
    if (det == 0) return FALSE; // parallel lines - no solution

    solution->x = ((l2.b*l1.c) - (l1.b*l2.c))/det;
    solution->y = ((l1.a*l2.c) - (l2.a*l1.c))/det;
    return TRUE;
}
```

Listing 4.6: Solving a linear equation system.

After getting the point of intersection between the two lines, we need to perform a last simple check to ensure that the point is on both the two segments, because it might be that the lines intersect somewhere else. This check is as simple as writing:
4.2.2 Considerations

So we have seen the code used to detect if a point is contained in a polygon or geographical area represented by it. In order to have a naive solution ready in no time, we could implement a linear scan of our areas’ database applying the point-in-polygon algorithm to every polygon-area we find. It is evident that while implementing advanced search features, a linear scan is an unfeasible solution. The way we will proceed will be by building an index over our data that will allow faster and acceptable times for retrieval of entries. In order to approach the problem, geospatial data structures were researched and will be presented in the next Literature section 2.

4.3 Spatial coordinates and projection systems

Before dwelling on the various software options in commerce, it is presented a brief explanation about the spatial coordinate systems (SRIDs) and projections types (such as WSG 84 from the EPSG open database).

4.3.1 SRID

The acronym SRID stands for Spatial Reference System Identifier. It is used to uniquely identify spatial coordinate systems. In most commercial tools (such as IBM DB2, Microsoft SQL Server, MySQL, Oracle, PostgreSQL) SRIDs are used to define which coordinate system is used by the spatial columns (column type GEOMETRY). Some SRID examples are:

- SRID 4326 is the common latitude/longitude coordinate system.
- SRID 3857 is used in Google Maps and Bing and is in meters.
- SRID 27700 is a local coordinate system called British National Grid (also in meters).

A SRID is characterized by a WKT string describing datum, geoid, coordinate system and map projection of the spatial objects.

With regards to our comparison about Elasticsearch and PostGis, we notice that Elasticsearch does not support multiple SRIDs, differently from PostGis. Elasticsearch always assumes SRID 4326, allowing only the latitude/longitude coordinates system.

PostGIS is much more flexible. To make an example in PostGis, here is the example definition of a table with a spatial column, as in the online documentation [4]:

```sql
CREATE TABLE mytable (
    id SERIAL PRIMARY KEY,
    geom GEOMETRY(Point, 26910),
    name VARCHAR(128)
)
Listing 4.7: Create table with geometry column.

The created table presents a spatial column for storing points, whose SRID is 26910, which is the Projected coordinate system for North America. It is common for spatial vendors to refer to the EPSG authority for the SRID implementation.

**EPSG**

The EPSG acronym stands for European Petroleum Survey Group and is a structured dataset of coordinate system and transformations. It is publicly available online for download or included in major spatial vendor softwares. Unfortunately the dataset cannot record all possible geodetic parameters around the world. EPSG codes are one implementation for SRIDs.
Chapter 5

Spatial data: commercial software

While chapter 3 focused on algorithms and data structures used for multidimensional datasets and geospatial search, the following chapter will outline the software solutions actually used in real world applications and provide some (quite hard to find) implementation details. Focus is on Elasticsearch and PostGis. All following softwares use any of the implementations mentioned in Chapter 3.

5.1 Solutions overview

We saw in section 4.2 that a linear scan over the list of geometries (to find all the ones where the point-in-polygon algorithm returns true) is an unfeasible and non-scalable solution (see results in table 8.1). An index over the geospatial data must be built. At this point an entire world of possibilities opens up. Since the first years of invention of geospatial data structures (in the 80s), many many tools and software have been created. We will try to give an overview here and compare some of them in order to eventually take a decision.

During the initial planning and after a preliminary research, a few entries compose the list of possible options to solve the problem of spatial indexing and searching:

1. Spatial Indexes and ST precise shapes functions provided by PostGIS.
2. MySQL 5.6 or higher in order to have the ST (precise shapes) functions.
3. MBR (Minimum Bounding Rectangle) features in MySQL 5.5 - search would need to be integrated with external python code (the Shapely library is perfect for the purpose).
4. ElasticSearch spatial search features (QuadTrees implementation).
5. Java Topology Suite, also known as JTS.

The rest of the chapter will focus on presenting some entries in the list, analyze their features and their pro et contra.
5.2 Library solutions

Here is the outline of the main library solutions for the language Python and Java.

5.2.1 Java Topology Suite

“JTS Topology Suite (JTS) is a Java class library providing fundamental geometric functions according to the geometry model defined by the OpenGIS Consortium” [20].

The JTS library seems perfect for the task at hand. It is licensed under GNU LGPL agreement and it complies to the OpenGIS Consortium. It is complete of a wide range of useful spatial functions (buffer, convex hull, MBR) and implementations. In fact, it comprises a static R-tree implementation and a specialized quadtree one. The library allows users to define the required precision. According to the official website from VIVID Solutions [29]:

“JTS is fast enough for production use”

Moreover the library is in a very mature state and the implementation is very robust. It would seem like the perfect tool for the job. Unfortunately, besides the fact that is written 100% in Java (company requirements are to use Python). we read that:

“there is no support for managing data on disk efficiently.”
(according to the external opinion from the Handbook of Data Structures [20]).

5.2.2 Python Toblerity project and Pyproj

The Toblerity project contains a few python libraries and projects used for manipulation and processing of geospatial data. Among them, we should pay attention specifically to Fiona and Shapely, because they are used in various stages of the current work, even though not as the final solution for the geospatial search features. Pyproj is not part of the Toblerity project but is used as well and hence is appearing here.

Fiona

Fiona is one of the best python libraries for interfacing with multi-layered GIS formats (especially old ones like the shapefile). Fiona will come very handy in the next chapter, during the building of the Reverse Geocoder. Fiona is able to integrate smoothly with Shapely and Pyproj. See chapter 6 for a batch example using Fiona.

Shapely

Shapely is the core library for this work. Shapely is a porting of the aforementioned JTS library and allows easy manipulation and analysis of geometric objects. Unfortunately, Shapely is able to handle objects only in the cartesian plane, reason why we also need Pyproj. To make some examples, Shapely is able to handle both WKT, WKB and GeoJson formats, but also operations
compliant to the OpenGIS Consortium Specifications. In the course of this work, Shapely was used to handle the data and check its validity, as explained in Chapter 7 in the section regarding data validity.

**Pyproj**

Pyproj is a straightforward porting of the Java library Proj4j (in turn porting of the C library Proj4). Proj4 is in active use by PostGIS. The library provides advanced cartographic projection functionalities. It is used to transform latitude/longitude coordinates to Cartesian coordinates for them to be handled by Shapely. Shapely is used to execute geometrical validity checks (see 7).

The rest of the softwares are database systems and will be allocated entire sections in this same chapter.

### 5.3 SQL DBMS: PostGres with PostGIS extension

PostGis is an extension for PostGres, the well-known open-source SQL relational database. It is one of the common SQL choices together with MySQL, Microsoft Server or Oracle SQL server. At the time of writing, PostGis happens to be one of the best solutions for geospatial data, meaning that it is the most robust and tested, has been around for the longest time, and it comprises the highest number of features among the competitors. The first PostGis version appeared in 2001, while the first stable 1.0 goes back to 2005, showing a certain maturity compared to other commercial tools.

According to David Smiley, few NoSQL solutions are able to feature solutions to handle geospatial data, most of which are very basic as of 2013. Things are different in the SQL world, where some quite advanced results have been achieved. As already outlined, PostGis is the leader in the field, featuring spatial indexing at an advanced level of maturity. In this context, “advanced maturity” of the tool means that the system allows indexing of complex shapes (as can be polygons) and geospatial search using complex shapes (like polygons) in the query (but also simple shapes like points). As an example, we can think of a use-case query of the type: “retrieve all polygons that intersect with this given polygon”. This problem is not trivial and it is actually handled by PostGis very well.

---


2 From David Smiley own LinkedIn page [https://www.linkedin.com/in/davidwsmiley](https://www.linkedin.com/in/davidwsmiley) about himself: “I’m an Apache Lucene/Solr expert, offering my skills as an independent consultant. [...] More broadly I’m a full-stack web developer with experience at all tiers from HTML/CSS/JavaScript up front, to Java, Grails, and Spring, to Lucene/Solr and SQL relational databases and a dabbling in some NoSQL stores. In addition, special interest areas of mine are spatial / geometric / geospatial algorithms, and multi-threaded / concurrent programming. This is in addition to search, of course”. David Smiley github page: [https://github.com/dsmiley](https://github.com/dsmiley)

3 See [https://www.youtube.com/watch?v=L2c0Gv0Rebs](https://www.youtube.com/watch?v=L2c0Gv0Rebs) for the 77 minutes long talk about spatial features in nowadays DBMS solutions
5.3.1 Features

Technically speaking, PostGIS uses R-tree-over-GiST (Generalised Search Tree) spatial indexes for high speed spatial querying. PostGIS is compliant with the Open Geospatial Consortium’s (OGC) OpenGIS Specifications (see the previous section 4.1 and references at [14]). All the geometry times documented in the OpenGIS specification are supported, among which: linestrings, polygons, multipoints, multilinestrings, multipolygons and geometrycollections. From the official postgis developer’s manual 2.1.5 [23]:

“PostGIS is compliant with the Open Geospatial Consortium’s (OGC) OpenGIS Specifications.”.

Data can be stored using a geometry type, both in WKT (Well-Known Text format) or WKB (Well-Known Binary format) 4.1.2.

Spatial operators choice is wide as well. It goes from operators for determining geospatial measurements like area, distance, length and perimeter to operators for determining geospatial set operations, like union or difference. Most importantly (specially because of intersect in this work), the spatial predicates are among the best around, in fact all kind of interactions are included: intersection, containment, within predicate and others. Please see [23] for a complete list.

PostGres in real world applications

PostGres is nowadays used by hundreds of companies [5]:

“[...]hundreds of companies who have built products, solutions, web sites and tools using the world’s most advanced open source database system [Postgres]”

Always according to the official site, a few remarkable companies among the considerable total are: Government agencies such as U.S. Agency for International Development, U.S. Centers For Disease Control and Prevention, U.S. Department of Labor, U.S. General Services Administration, U.S. State Department, IMDB.com, The Internet Movie Database, SourceForge, ADP Dealer Services (article), Courage To Change, Safeway, Tsutaya, The Rockport Company, LLC, Technology, Apple (article), Fujitsu, Red Hat, Sun Microsystems, Cisco, Juniper Networks (documentation), Skype.

5.4 ElasticSearch

ElasticSearch is a very recent key-value solution based on Lucene full-text search engine. While Lucene is relatively old in terms of internet years (1999) and therefore well proven and tested, ElasticSearch first version was released only in 2010. When saying that ElasticSearch is based on Lucene, it is actually meant that ElasticSearch is Lucene, with added a layer of new functionalities for improved scalability which make use of the Lucene Java libraries. So the main reason for the great success that ElasticSearch earned in only four years is the extremely high scalability that brings to the table. This is achieved thanks
Figure 5.1: ElasticSearch general architecture from www.elasticsearch.org

to an additional layer which exploits distributed computing techniques for parallelizing search operations. See Section 5.4.1 for details about ElasticSearch architecture.

5.4.1 Architecture

First of all let us introduce some terminology about ElasticSearch internal distributed structure. Please refer to Figure 5.1.

1. **node**: a node is a running instance of ElasticSearch. Nodes belong to clusters. A unicast or multicast mechanism is used at startup by a node to find which cluster it belongs to (a name can be assigned to each node and it will be looking for nodes with the same name to form a cluster). It is good practice to start up a node per server.

2. **cluster**: a cluster is a set of nodes which share the same cluster name as explained above. Each cluster chooses a node to be the master node. If that node fails, then another one is chosen to be master.

3. **shard**: in order to understand the shard, we need to explain the ElasticSearch index. In practice, an index is the ElasticSearch equivalent of a SQL table. In other words it is the **logical** object where the data for a specific application resides. This being said, a shard is a single Lucene instance (we said that ElasticSearch uses Lucene as its core and focuses on managing multiple Lucene instances). Each index can be assigned any number of shards (default is five). The data inside the index will be distributed among the various Lucene instances, allowing so distribution of the work. In fact, the shards can be assigned to different nodes, each one of them searching in parallel over different portions of the index. ElasticSearch is clever enough to be capable of intelligently assigning shards to nodes as needed (depending on how many nodes are running). Shards can be replicated for achieving better reliability in case of node failure. Therefore, we can have **primary shards** and **replica shards**. In other words, the index is the logical object we should refer to when developing. The shards are the implementation details on how to maintain the index.

In Figure 5.1, it is possible to visualize what just introduced about ElasticSearch architectural structure. The picture represents a cluster composed
Figure 5.2: Cluster composed by a single node *Gaia* which contains five shards.

by three nodes (the big boxes). Each node contains shards. How to distribute the shards is handled transparently by ElasticSearch. In the figure we can see that each node is assigned two shards. The three green shards are primary shards which standalone form an index, while the three grey ones are the replica shards. The shards are cleverly distributed by ElasticSearch so that in case of failure of one node, no data loss is experienced.

A practical example follows, including figures to understand the cluster-node-shards architecture. The pictures can be obtained using a monitoring plugin for ElasticSearch called *Head*.

Let us begin by firing up an instance of ElasticSearch. What happens? The new node uses multicast to search a cluster having the same name.cluster. All nodes with the same value belong to the same cluster. In this case, being the node the first one, it will not find any cluster so it will create a new one. The index created in the picture is used for storing meta-information about the *Marvel* plugin. The default number of shards is five. This number can be changed. The situation is depicted in Figure 5.2. We can notice the index *Gaia* composed by the five green primary shards numbered from zero to four.

We can also notice five unassigned shards. These are the replica shards which cannot be assigned because there is no other node running and it makes no sense to put the replica shards on the same node of the primary shards. The replica shards are in fact used for replication of data and their purpose is to have a backup in case of hardware failure, or for incrementing read speed. For this reason, ElasticSearch always separates the primary shards from the replica shard putting them in different nodes. We remind that each node should be started up on a different machine.

Last thing worth noticing is the indicator for the cluster health. The state of the cluster is indicated as yellow (to the top on the right). This means that all the primary shards are active and working but not all the replica shards are. In fact, we do not have any replica shard active at all.

The cluster health color might be one of the following:

1. **GREEN**: all primary and replica shards are active.

2. **YELLOW**: all primary shards are active but not all replica ones are.
Figure 5.3: Cluster composed by two nodes *Gaia* and *Joseph* which contain five shards each (primary and replica).

Figure 5.4: Cluster composed by three nodes *Gaia*, *Joseph* and *Ricochet* with a clever distribution of shards.

3. **RED**: not all primary shards are active.

Every time we index a document in an ElasticSearch index, it intelligently assigns it to one of the index shards, choosing it in a manner such that the data remains should remain balanced among the shards. Now we can see what happens in case we start up another ElasticSearch instance (another node) in Figure 5.3.

The replica shards were cleverly assigned to the new node by ElasticSearch. Moreover, we can see that the cluster health became green. Now in case we introduce a third node we are presented with Figure 5.4.

As clearly evident, ElasticSearch is able to distribute the shards among the nodes, always making sure to keep the primaries (bold squares in the picture)
and the replicas on different nodes.

5.4.2 ElasticSearch set up

A brief introduction on ElasticSearch terminology and inside functionality before describing our use case:

1. ElasticSearch basic indexed unit are called Documents. The equivalent of the classic Table is called Index. ElasticSearch is schema-free but a schema can be defined and is called Mapping.

2. ElasticSearch is based on Lucene, and it uses Lucene for search. Lucene is a full text search engine so the basic mechanism for indexing the data is to build an inverted index that map words (strings) to the documents they are found in. To make an example related to our case, in order to index spatial shapes Lucene computes the geohash of the shapes (to get to the case of a mapping from string to document).

3. ElasticSearch has a friendly web CRUD API for interfacing with the outside. It is well documented at http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/docs.html. It is very easy to execute any API operation by using the “curl” utility.

CRUD vs. REST API

As previously stated, ElasticSearch implements a CRUD API, conceptually different to the REST API. Understanding the difference between the two requires capacity of abstraction being it slight.

CRUD stands for Create, Read, Update and Delete, which are the main relational database operations. CRUD is used when the basic operations involve a data-store. The API is used to directly manipulate data records. The data records and tables are passive and the API can act on them.

REST (Representational State Transfer) is a set of guidelines used to manage complex systems using the simple and wide-spread protocol HTTP. REST makes use of verbs and names in order to operate on resources. The resources are identified by URLs. The resources are not necessarily only data records but can be complex abstractions of objects.

The verbs used are simply HTTPS verbs, an example list is:

1. GET — it is used to request a representation of a resource. The only effect should be retrieving data.
2. POST — it is used to create a resource.
3. PUT — it is used similarly as POST. The resource gets created or updated if already existing.
4. DELETE — deletes an existing resource.

For example the REST API could be used for manipulating user data at a higher level of abstraction than CRUD. Manipulation of data often involves more than simply creating or deleting rows from a database. Data may be processed, validated or go through a certain number of steps before hitting a data-store.

42
In conclusion, REST is used in complex systems such as web services in order to interact at a high-level with the entire system. CRUD instead is more primitive, being it used mostly for simple database operations on passive data records.

Example of REST APIs are the Google Glass API (Mirror API), Twitter APIs, Flickr APIs. An example of CRUD API is obviously ElasticSearch as mentioned above.

**Cluster set-up**

ElasticSearch is nothing more than a distributed scalable layer built on top of Lucene. It allows scalability when number of queries gets really high, a problem well felt in 2014, the era of the new web 2.0. In order to achieve low response times for an extremely high amount of queries over databases with millions of records, the task need to be parallelized. Being this section far from an introduction on the various techniques for tasks parallelization, we will just say that ElasticSearch exploits a distributed architecture to handle the huge amount of work. In practice, clusters need to be configured and set up, according to user needs. An ElasticSearch cluster is a set of nodes that are capable of interacting with each other (using their name to find each other inside the network). They practically achieve the powerful scalability presented by this tool.

As a first step after downloading the developer version, we can start the cluster and test if it is working:

```bash
$ ./bin/elasticsearch
$ curl 'http://localhost:9200/?pretty'
{
  "status" : 200,
  "name" : "Illyana Rasputin",
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "1.4.2",
    "build_hash" : "927caff6f05403e936c20bf4529f144f0c89fd8c",
    "build_timestamp" : "2014-12-16T14:11:12Z",
    "build_snapshot" : false,
    "lucene_version" : "4.10.2"
  },
  "tagline" : "You Know, for Search"
}
```

From the output it is noticeable the involvement of Lucene (field “lucene_version”) as the core which ElasticSearch is based on.

**ElasticSearch Mapping**

ElasticSearch is well known for being a schema-free database system. The “elasticity” implied in the naming of this tool regards this aspect as well. There is no constraint to define a schema in advance, in fact ElasticSearch is smart enough to infer a type for each field of the documents inserted. Nevertheless,
it is possible to specify the schema if needed. It is actually highly suggested, both for documentation purposes and as a point of reference but also to avoid misinterpretation on the engine side. It is worth saying that ElasticSearch will continue to accept any kind of documents though, regardless of their schema. In ElasticSearch terminology, a schema is referred to as a mapping. Another important thing worth noticing is that in ElasticSearch there is no concept of “relations” as intended in SQL tools. In other words, it means that every document is stored in the same place. We would say in the same “table” if we were treating a SQL tool. For ElasticSearch instead, it is more appropriate to refer to this structure as to documents which live in an index and it is made retrievable through search over the fields as specified in the mapping (if defined in advance), or as inferred by the engine itself (if the mapping is not defined).

In order to define a mapping it is sufficient to use the appropriate endpoint, described in table 5.1, and to send the correct information to ElasticSearch in JSON format (an example in listing 6.4)

```json
{
    'properties': {
        'information': {
            'type': 'string',
            'store': True,
            'index': 'not_analyzed'
        }
    }
}
```

Listing 5.1: Mapping documents into ElasticSearch.

**ElasticSearch Insert Data**

Data can be inserted hitting the endpoint described in 5.2
Table 5.3: Elasticsearch query endpoints API.

For our purposes, we will be using a JSON document of the following type:

```json
{
    "_source": {
        'zipcode': string,
        'geometry': <GeoJson>,
    }
}
```

Listing 5.2: Example JSON document to be inserted into ElasticSearch.

**ElasticSearch Query**

ElasticSearch query mechanism is quite complex. There exist *queries* and *filters*. In general, a query should be used in the cases:

1. full text search
2. when we are interested in sorting results based on a relevance score.

while a filter should be used:

1. binary yes/no searches
2. for querying on exact values.

Queries can be performed by using the endpoints described in 5.3. There is the option of using JSON format or text parameter as in normal GET operations.

Our case is the second in table 5.3. The query used by this implementation is shown in listing 5.3.

```json
{
    "query": {
        "filtered": {
            "query": {
                "match_all": {}
            },
            "filter": {
                "match": {}
            }
        }
    }
}
```
Listing 5.3: Query to retrieve shapes that contain a certain point.

5.4.3 ElasticSearch strengths and defects

A CRUD API is available for interfacing with ElasticSearch. Though ElasticSearch scales outstandingly well thanks to its distributed architecture, it should not be the first choice over the relational databases in case some of the following reasons are considered important:

1. ElasticSearch is slow for any kind of grouping queries (GROUP BY queries, to speak the SQL language), count queries, sum of values and so on.

2. ElasticSearch is not good at executing different operations with transaction support and rollback support.

ElasticSearch is very good for a list of other tasks:

1. Storing great amount of JSON data.

2. Take advantage of multiple machines to speed up search operations (shards and clusters can be easily configured).

3. Full-text search.

4. ElasticSearch is schema-free. This means that schema need not be defined. It is possible to start indexing without defining any schema, augmenting flexibility in case changes are needed after insertion.

Most of the previous drawbacks and advantages applies for ElasticSearch, as well as Lucene, Solr and any other full-text based search engine.

To summarize, the killer-feature of ElasticSearch is its incredible flexibility. It seems that its name was well chosen. It really is an “elastic” tool, needless of a schema, able to scale and distribute workload according to necessity.

5.4.4 ElasticSearch used in real world applications

Some examples of real world applications where ElasticSearch is used⁴:

⁴From http://www.elasticsearch.org/
“Wikipedia uses Elasticsearch to provide full-text search with highlighted search snippets, and search-as-you-type and did-you-mean suggestions.

The Guardian uses Elasticsearch to combine visitor logs with social-network data to provide real-time feedback to its editors about the public’s response to new articles.

Stack Overflow combines full-text search with geolocation queries and uses more-like-this to find related questions and answers.

GitHub uses Elasticsearch to query 130 billion lines of code.”

**ElasticSearch geospatial indexing implementation**

ElasticSearch offers two different kinds of implementation for spatial indexing. They can actually be specified by the user, including some of the parameters. When creating a mapping of “Geo Shape” type, an option “tree” can be passed in order for it to use Quadtrees or Geohash for the underlying PrefixTree. Some options that can be set are the tree levels and the precision we would like to maintain. We need to be very careful in setting these parameters because a little mistake could cause problems regarding either performance and space (too many tree levels that we do not actually need) or precision (too few levels that allow us to save space and time at the cost of accuracy). A procedure for correctly tuning these parameters will be shown in section 6.2.1
Chapter 6

Reverse Geocoder and experiments

The following chapter follows the engineering process to develop the application that goes under the name of Reverse Geocoder. It is the inverse use-case of what we are used to in everyday life. Instead of searching for an address on a map, we search for a point on a map in order to get its address. Although the normal user is usually interested in finding out the directions to get from a point A to a point B (service offered for example by Google Maps or Bing Maps), nowadays there is also great interest in the reverse geocoding process. In fact most of the “Apps” on our mobiles rely on reverse geocoding. Social applications such as Trip Advisor, Google Maps, Facebook or Whatsapp. They all apply reverse geocoding in order to allow the user to tag its data with the location produced (pictures or instant messages) or to display an address to the user so that he can orient himself. Let’s keep in mind that our phone has only a pair of coordinates as input from the GPS dongle! So here experiments and tests are performed and timings taken while going through the steps for creating a reverse geocoding software application.

6.1 Shapefiles

The shapefile is a widespread file format to represent GIS data. Historically it was developed by Esri[10] (by which it is regulated). The shapefile is able to represent geometries (points, lines, polygons) and attributes that are associated with them. For example the geometries could be used to represent geographical objects like rivers, lakes, tax parcel areas, zipcodes, national park’s areas and so on. To make a practical example, we can have a shapefile with geometric information associated to zipcodes, meaning that each geometry will have a correlated zipcode as an attribute in the shapefile. In the United States a great quantity of datasets are made public and the zipcodes geographical information is among those. The entity responsible for these datasets is the United States Census Bureau [3].

Technically, the shapefile is composed by some mandatory files and some additional files. Among the most important:
Figure 6.1: Geometries associated to US zipcodes.

- `.shp` - this file contains the geometries themselves.
- `.shx` - this file contains a positional index for the geometries.
- `.dbf` - this file contains the attributes. It is organized in columns. There are value for each shape. The format is dBase, a very old database management system appeared first in 1979 [17].
- `.prj` - this file can contain the projection information for the shapes. In practice it tells us which coordinate system we should use for the geometries. Format used is WKT (see 4.1.2 for explanation of WKT and WKB).

6.1.1 Process Shapefiles

After describing the format of a shapefile, we now will see how to handle it. There exist dozens of tools to visualize shapefiles. A common choice for Linux users is QuantumGIS (QGIS). It allows instant visualization of the geometries (see Figure 6.1.), and a nice tabular visualization of all the features associated to the geometries (see Figure 6.2 for an example). There are many more advanced features (connection to databases like PostGIS or MySQL, geoprocessing tools, shapefile editing tools and so on). The possibility to connect to PostGIS via a GUI might be useful later for our purposes.

Since the shapefile is an ancient format and it hardly well interacts with modern database systems, a batch converting the shapefile to the modern GeoJSON format is introduced. The batch makes use of the python library called Fiona, used for geospatial applications and saves the information in json format (see line 22).

```python
import sys
```
2 import fiona
3 import json
4 from shapely.geometry import shape
5
6 def extract_shapefile(shapefile_filename, output_filename):
7     c = fiona.open(shapefile_filename)
8     i = 0
9     database = []
10    for record in c:
11        element = {
12            'properties': {
13                'zipcode': record['properties']['ZCTA5CE10']
14            }
15        }
16        element['geometry'] = record['geometry']
17        database.append(element)
18
Figure 6.2: Features associated to US zipcodes geometries.
### 6.1.2 Dataset and conditions for the experiments

The reverse geocoder project makes use of a dataset downloaded from the United State Census Bureau [3] that includes all United States zipcodes and their geometries. The dataset is also used for experiments and benchmarks to compare different proposed options and solutions. All experiments are meant under the same conditions using the same dataset and hardware.

The physical machine’s hardware is listed in table 6.1.

### 6.2 ElasticSearch benchmarks

Let’s get started with elasticsearch. The tests exploit the spatial indexes naturally offered by the engine which use an underlying quadtree implementation.

ElasticSearch great advantages come from the ability to distribute tasks. It makes sense to exploit this feature, also considering that a multi-core machine is available. Conditions consist in two instances of an ElasticSearch cluster for a total of 10 shards (Lucene underlying instances).

```
curl -L -O http://download.elasticsearch.org/PATH/TO/VERSION.zip
unzip elasticsearch-$VERSION.zip
```

```bash
cd elasticsearch-$VERSION
```

52
6.2.1 Elasticsearch precision parameter tuning

We recall from chapter 5 that Elasticsearch allows some configuration for indexing spatial data. In particular, it is possible to specify either the precision parameter or the tree_levels. In order to correctly engineer our application, it might be useful to know how the two values relate to each other. Given that it is nearly impossible to find any information about this anywhere except on a couple amateur blogs online, the relation at hand should be investigated. Starting from the official Elasticsearch code 6.2, public on Github [2].

```java
/** Earth ellipsoid major axis defined by WGS 84 in meters */
public static final double EARTH_SEMI_MAJOR_AXIS = 6378137.0; // meters (WGS 84)

/** Earth ellipsoid minor axis defined by WGS 84 in meters */
public static final double EARTH_SEMI_MINOR_AXIS = 6356752.314245; // meters (WGS 84)

/** Earth ellipsoid equator length in meters */
public static final double EARTH_EQUATOR = 2*Math.PI * EARTH_SEMI_MAJOR_AXIS;

/** Earth ellipsoid polar distance in meters */
public static final double EARTH_POLAR_DISTANCE = Math.PI * EARTH_SEMI_MINOR_AXIS;

/**
 * Calculate the number of levels needed for a specific precision. Quadtree
 * cells will not exceed the specified size (diagonal) of the precision.
 * @param meters Maximum size of cells in meters (must greater than zero)
 * @return levels need to achieve precision
 */
public static int quadTreeLevelsForPrecision(double meters) {
    // cell ratio
```
double ratio = 1+(EARTH_POLAR_DISTANCE / EARTH_EQUATOR);
// convert to cell width
double width =
    Math.sqrt((meters*meters)/(ratio*ratio));
long part =
    Math.round(Math.ceil(EARTH_EQUATOR / width));

// (log_2)
int level = Long.SIZE -
    Long.numberOfLeadingZeros(part)-1;
// adjust level
return (part<=(1l<<level)) ?level :(level+1);
}

Listing 6.2: Elasticsearch code for computing levels.

At this point a very quick snippet of code using the information attained will allow us to have a gross overview over the relation between the two parameters (see 3.2.2 for reference).

public static void main(String[] args){

    int level, last_level = 1;
    for (double meters=100000000; meters>0; meters*=0.999){
        level = quadTreeLevelsForPrecision(meters);
        if (level > 50) return;
        if (level != last_level){
            System.out.println(meters + "&" + level);
            last_level = level;
        }
    }

Listing 6.3: Reverse engineer levels from precision.

The results are shown in table 6.2. The most interesting values are around level 25. We may notice how we need 23 levels in order to achieve a precision of about 10 meters. Thanks to the table, we can settle for a still perfectly acceptable precision of 15 meters gaining the benefits of a level less in the tree (precision 15 is achievable with 22 levels). The goal is to find the optimum value where the needs of acceptable precision and minimization of levels are met. We are interested in minimizing the levels because all algorithms on trees are affected by the tree’s height, as seen in chapter 2.
<table>
<thead>
<tr>
<th>Precision</th>
<th>Levels</th>
<th>Precision (continue)</th>
<th>Levels (continue)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.0034236982680425E7</td>
<td>1</td>
<td>1.7891842615170492</td>
<td>26</td>
</tr>
<tr>
<td>3.001112915453931E7</td>
<td>2</td>
<td>0.8944136321603932</td>
<td>27</td>
</tr>
<tr>
<td>1.5002570506397454E7</td>
<td>3</td>
<td>0.44711758458910683</td>
<td>28</td>
</tr>
<tr>
<td>7499788.516467171</td>
<td>4</td>
<td>0.22351418545121968</td>
<td>29</td>
</tr>
<tr>
<td>3749146.039190282</td>
<td>5</td>
<td>0.11173479375416046</td>
<td>30</td>
</tr>
<tr>
<td>1876075.05979602</td>
<td>6</td>
<td>0.055912161677798</td>
<td>31</td>
</tr>
<tr>
<td>937850.3626005725</td>
<td>7</td>
<td>0.027950502792382967</td>
<td>32</td>
</tr>
<tr>
<td>468831.616328482</td>
<td>8</td>
<td>0.013972462904434173</td>
<td>33</td>
</tr>
<tr>
<td>234369.0350128818</td>
<td>9</td>
<td>0.006984837484533265</td>
<td>34</td>
</tr>
<tr>
<td>117161.13559710122</td>
<td>10</td>
<td>0.003491721897494456</td>
<td>35</td>
</tr>
<tr>
<td>58627.50668326837</td>
<td>11</td>
<td>0.001747259855737827</td>
<td>36</td>
</tr>
<tr>
<td>29307.904347777145</td>
<td>12</td>
<td>8.734556118738934E-4</td>
<td>37</td>
</tr>
<tr>
<td>14651.028260486952</td>
<td>13</td>
<td>4.36640665330134E-4</td>
<td>38</td>
</tr>
<tr>
<td>7324.052465247897</td>
<td>14</td>
<td>2.182767710552737E-4</td>
<td>39</td>
</tr>
<tr>
<td>3661.2955459496925</td>
<td>15</td>
<td>1.0911660906867954E-4</td>
<td>40</td>
</tr>
<tr>
<td>1832.114617152077</td>
<td>16</td>
<td>5.4602020497895167E-5</td>
<td>41</td>
</tr>
<tr>
<td>915.8745270158553</td>
<td>17</td>
<td>2.72955628591379E-5</td>
<td>42</td>
</tr>
<tr>
<td>457.84589096310305</td>
<td>18</td>
<td>1.3645058278124848E-5</td>
<td>43</td>
</tr>
<tr>
<td>228.8772683249531</td>
<td>19</td>
<td>6.821167835016522E-6</td>
<td>44</td>
</tr>
<tr>
<td>114.41580014117493</td>
<td>20</td>
<td>3.4099034012962948E-6</td>
<td>45</td>
</tr>
<tr>
<td>57.25373970755796</td>
<td>21</td>
<td>1.7063178282623635E-6</td>
<td>46</td>
</tr>
<tr>
<td>28.621157599081066</td>
<td>22</td>
<td>8.529886827319565E-7</td>
<td>47</td>
</tr>
<tr>
<td>14.307723399670222</td>
<td>23</td>
<td>4.264092464956063E-7</td>
<td>48</td>
</tr>
<tr>
<td>7.152434284769923</td>
<td>24</td>
<td>2.1316208045647501E-7</td>
<td>49</td>
</tr>
<tr>
<td>3.575503577258937</td>
<td>25</td>
<td>1.0655977403818975E-7</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 6.2: Precision and Tree Level relation.
6.2.2 Elasticsearch bulk insertion test

After extracting the dataset from the old shapefile format to Geojson (easy to process with python, PostGis and Elasticsearch), it can be ingested into Elasticsearch. Before ingestion, the setup used for the Elasticsearch index mapping is shown in listing 6.4.

```json
{
    
    '_all': {
        'enabled': False
    },
    
    '_source': {
        'enabled': False
    },
    
    '_timestamp': {
        'enabled': True
    },
    
    'dynamic': 'strict',
    
    'properties': {
        
        'information': {
            'type': 'string',
            'store': True,
            'index': 'not_analyzed'
        },
        
        'zipcode': {
            'type': 'string',
            'store': True,
            'index': 'not_analyzed'
        },
        
        'geometry': {
            'type': 'geo_shape',
            'precision': '15m'
        }
    }
}
```

Listing 6.4: Mapping documents into ElasticSearch.

Insertion is operated in bulk, 500 entries per time. Entries look like listing 6.5. Elasticsearch configuration is 1 cluster with 2 nodes running and 10 shards in total (5 primary and 5 replica) as in table 6.3. The insertion can be performed using the python API for Elasticsearch (elasticsearch-py). The entire API is documented at [9].
<table>
<thead>
<tr>
<th>Clusters</th>
<th>Nodes</th>
<th>Shards</th>
<th>Primary</th>
<th>Replica</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3: Elasticsearch Configuration

```
"_index": string,
"_type": string,
"_id": integer,
"_source": {
  'zipcode': string,
  'geometry': GeoJson,
}
```

Listing 6.5: Document to insert into Elasticsearch.

Preliminary results are shown in Figure 6.3. The expected time complexity of the algorithm is $O(n \cdot \log n)$ and is plotted in the figure. As we can see, the results seem reliable. They will be compared with results obtained by the other tools ahead in the dissertation.

**Elasticsearch query test**

The query speed test results are shown in the related section 8.1.2 and are computed as an average of 500 queries.

### 6.3 PostGis benchmarks

Postgis is easily configured on a linux system.

```
$ sudo -u postgres psql postgres
$ \password postgres

Let's setup a new user:

sudo -u postgres createuser --superuser $USER
sudo -u postgres psql
postges=# \password $USER
sudo -u postgres createdb $USER

This way logging in will be as easy as typing psql. Database creation:

sudo -u postgres createdb zipcodes_db
```
We can easily create a table with a spatial column of type GEOMETRY. The zipcodes in the US are unique and can be primary keys.

```
CREATE TABLE zipcodes_table (
    zipcode VARCHAR(6) PRIMARY KEY,
    shape GEOMETRY (GEOMETRY, 4326),
);
```

### 6.3.1 PostGIS bulk ingestion

In order to perform the bulk ingestion, the following code can be inserted in a loop over the dataset for ingesting records into the DBMS. The zipcode '94105' is an example while the text representation of a GeoJSON shape should be inserted where indicated.

```
INSERT INTO zipcodes_table (zipcode, shape) VALUES
    ('94105', ST_GeomFromGeoJSON('<GeoJSON>'))
```
It is better to add the spatial index after the ingestion is complete. Doing so allows for better ingestion times. If we ingest after index creation, the DBMS needs to update the index after each INSERT operation.

```
CREATE INDEX zipcodes_index ON zipcodes_table USING GIST (shape);
```

Finally we can achieve our original purpose, execute a query to find all the shapes containing a certain point. The following is an example with a point situated in the city of San Francisco. The result '94105' is correctly returned.

```
SELECT zipcode FROM zipcodes_table
WHERE ST_Contains(shape, 'POINT(-122.400475,37.787921)');
```

We can see the results of the PostGIS ingestion in Figure 6.4 which represent time taken over number of records inserted. The figure presents a two-scales chart useful to show the behaviour of the algorithm. As expected the ingestion is characterized by $O(n \cdot \log n)$ complexity.

**PostGIS query results**

As seen for the Elasticsearch query test, the PostGIS experiment was computed as an average of 500 queries over the zipcodes dataset and results are shown in 8.1.2
Figure 6.4: PostGis ingestion times for the zipcodes database.
Chapter 7

Service Implementation

This chapter is dedicated to the practical implementation steps for data pre-processing and design of the User Interface solution. The sections will be treated in order, going from the data quality assessment and validation to the visualization solution. To the choice of validation criteria. To the conclusions drawn from the experiments phase for choosing the back-end tools for search. And finally to the set up of a Python web service.

7.1 Data Quality Assessment

The first step before starting the development of a software application is to properly understand the data at hand and its format. After clearing up the air about the format of the data, another important step is to verify its validity. In case of invalid data, it may be worth considering the insertion of a data validation step when collecting it.

So, an introductory question might be: what kind of data are we dealing with? The main goal of this phase is to get a better understanding of the dataset so to be able to plan accordingly how to proceed (mainly in terms of tools to utilize) and how to plan and design the system.

For the purpose of the current application, we will have a dataset where each data point represents a business. Each business has a “service area” associated to it, where food-delivery is guaranteed. The service areas can be expressed under different formats:

1. **Circle**: a Point (center) and a distance (radius) are specified. They identify a polygon in a 2D space.
2. **Polygon**: the Polygon is described as a list of coordinates.
3. **Zipcode**: a postal code describes a geographical area. It makes sense to use this notation in case the application is not purely geometric but has geographical meaning.

Circle and zipcodes are easy to deal with, from a programmer point of view. Likely everybody of us will be familiar with the international standard of the five digits postal codes, since we all have to deal with them when sending or receiving letters and packages. It should be trivial to get an idea of circles as well, because they are only composed by a point and a distance (the radius).
But how to deal with polygons? Any existing representation of polygons comprehends a long list of coordinate points. Any programmer can be overwhelmed by the sight of such a list, without being able to know if they make sense when plotted on a geographical map. For this reason it is presented the development of a specific tool used for visualization of geospatial data.

**First goofy approach**

A first attempt to understand the data may be to print it to screen. Getting in output a huge list of coordinates composing a polygon is not very helpful for a developer.

```
(-70.36471999999999, 43.114314),
(-70.36369599999999, 43.115557),
(-70.364682, 43.117054999999997),
(-70.364745, 43.119115999999998),
(-70.363068, 43.119712999999997),
(-70.361409, 43.123051),
(-70.35945199999999, 43.124737999999997),
(-70.359901, 43.12682),
(-70.358059, 43.130132),
(-70.357948, 43.132091),
(-70.351997, 43.132291)
[continuing...]
```

Something should be done for data visualization. Goal is to go from the raw coordinate output to something like Figure 7.1 where the zipcode 07030 of the United States is shown. Figure 7.1 was obtained with the visualization tool implemented.

7.1.1 **Google Maps API v3**

When facing the problem of visualizing 2D geographical data, among various options such as QuantumGIS (free on linux), ESRI ArcGIS (proprietary), Grass GIS (free software from the OpenGIS Foundation), there is the modern Google Maps API that is a quite obvious, fast and well-tested solution. The Google Maps Javascript API are nowadays in a very mature state (they just reached version 3), they are very easy to use and extremely powerful. Moreover, building a tool using Javascript API implicitly brings portability. The map can be visualized in any browser present on any computer system able to connect to the internet.

The Google Maps Javascript API are very powerful as they make available a wide range of features for map visualization. In order to use them, we just need to add the reference to the Google javascript code in our html file:

```
<script
type="text/javascript"
src="http://maps.google.com/maps/api/js?sensor=false">
</script>
```
It is worth noticing that from version 3, Google does not require to register a key anymore. Including the previous script is all that is needed for developing purposes. Creating a simple page with a map inside is easy as adding a \textit{div} (inside which we can draw):

\begin{verbatim}
<div class="" id="canvas" style="width: 500px; height: 400px;"> </div>
\end{verbatim}

and actually drawing inside it:

\begin{verbatim}
function drawPolygon(center_lat, center_lon,
                    lat_list, lon_list) {
    var mapOptions = {
        zoom: 10,
        center: new google.maps.LatLng(center_lat,
                                        center_lon)
    };
\end{verbatim}
var map = new
google.maps.Map(document.getElementById('canvas'),
mapOptions);

var polygonCoords = [];
for (var i in lat_list){
polygonCoords.push(new
googleg.maps.LatLng(lat_list[i],
lon_list[i]));
}

var polygon = new google.maps.Polygon({
paths: polygonCoords,
strokeColor: '#FFFF00',
strokeOpacity: 0.8,
strokeWeight: 2,
fillColor: '#FFFF00',
fillOpacity: 0.10
});

polygon.setMap(map);
}
Listing 7.1: Drawing polygons with Google Maps API v3.

In the code we can notice various details. It is possible to create a map giving as arguments some options and the HTML div where to draw. The options format is of the kind key:value:

var mapOpt = {
  property:value,
  property:value,
  ...
};

It is possible to specify dozens of options, such as the initial zoom level, where the map should initially be centered, which kind of map it should be (topographical, terrain etc.).

In order to draw objects on the map we first need to create the objects such as we are doing for the Polygon in the code. Secondly, every object must be added to the map. It is interesting to notice that the adding method is owned by the drawn object class, not by the map. This way it is simple to keep a list of past drawn objects and clear them from the map with a “clear” method instead of having to find a way to indicate to the map object which shapes we want to clear.

There exist many possible objects that can be drawn, many of which can find correspondence in the OpenGIS specifications from document [6].

To report an example from our code, we can analyze the creation of a Polygon. As seen previously for the map, we need to pass some options to
define the polygon. There are dozens of possible options. Some of the most important include the path, which is a simple list of `google.maps.LatLng` objects representing the list of coordinates that define the Polygon, the color of the external boundary, the internal color and opacity and so on.

For more details, it is possible to consult the official Google Maps API documentation [12]

**Quantitative audit**

Additional information that might be useful is a quantitative audit over the data. We could count how many data points present each of the data types or different combinations of them (with 3 different data types there are up to $2^3 = 8$ possibilities). It is good practice collecting all this information because it improves observability over the system becoming extremely helpful during bug solving or emergencies. Moreover, a quantitative audit of this type might be useful to decide in which direction to proceed. In this case, given the high majority of polygon data (from 60% to 90% in some cases), it is evident that it is better to focus the efforts on building a system with polygon data in mind, being circles and zipcodes less relevant.

**7.2 Data validation**

The second question to answer before starting using the data regards its validity. Is the data “good”? After understanding what kind of data we are dealing with, it is of utmost importance that the data is validated. In fact, a goal to keep in mind during development is the user experience, a concept that engineers sometimes forget (fortunately there are Product Managers that remind them). Basically, we need the user to have the best search experience possible. It must be avoided the case of presenting wrong entries in the search results because this would lead to rejection during the processing phase and thus, to a negative transaction experience for the user, with potential loss of the same users. In order to do so, data used by the search engine must be reliable.

**Zipcodes and circles**

The task of validation is trivial for zipcodes and circles. For a zipcode we could decide to check that every one of the five characters be a number between 0 and 9, or we could cross check with a national postal code database if we want to be more precise. Depending on how much we value the validation we can tune the effort put into this task. Every coordinate value such as the circle center can be validated making sure that it is a float number between -180 and 180 for the longitude and between -90 and 90 for the latitude. With regards to the circle, we may do a simple float validation for the “radius” field.

**Polygons**

Fortunately circles and zipcodes are human-readable and easily understandable after a quick look at them. Polygons are trickier. Polygons can be invalid in many possible ways, it all depends on which checks we are willing to apply or
which ideas we can come up with for a broken-data-situation. Unfortunately, for data of this nature, it is very hard to spot unexpected bugs or edge case situations. It is not simple to imagine edge cases with multidimensional data! The same OpenGIS consortium presents many cases where a polygon is invalid (see [6]). In order to solve this problem, heuristics are created, together with the visual techniques mentioned earlier.

7.2.1 Heuristics

There must be decided criteria for invalid data with regards to the application at hand. We will list here some of these criteria or heuristics for data validation. We can consider invalid any area that fails geometrical correctness check based on the following heuristics:

1. **Polygons invalid**: this case checks validity compliantly to the criteria specified in [6]. To make an example, a polygon whose own edges overlap raises an error. A visual example is in Figure 7.2

2. **Insufficient points for a polygon**: when we have a list of less than three points.

3. **Polygon void**: when we have a void list of points or a null pointer.

4. **Intersecting Polygons**: this case can be included depending on which policy to adopt in the application. After an experimenting time period, the decision from the business intelligence was to exclude it.

All these check can be easily enforced by using existing libraries like the Java Topology Suite in Java, or Shapely from the Toblerity project in Python. To make an example, in Python it is possible to use the Shapely library. Geometries have a `is_valid()` method that performs checks as stated in the specifications at [6]. Citing from the shapely online manual at [http://toblerity.org/shapely/manual.html](http://toblerity.org/shapely/manual.html):

```python
object.is_valid
Returns True if a feature is valid in the sense of
```

Quantitative audit

Once again, it may be worth executing a quantitative audit to find out the amount of “broken” data. The audit can be performed using techniques explained in last section. Thanks to this metrics, the observability over the system improves allowing to act for improving data quality (mainly by requiring the businesses to fix the data and re-ingest it). During the arc of time of the project, data quality has improved from roughly 72% on average to roughly 97% of good data after heuristics. Experiment run on live software executions have shown to be agreeing with the data, going from roughly 85% to 98%. Heuristics design is designed to be as “safe” as possible so that preliminary results are worse than the real ones.

---

7.3 Set up a python web service

At this point in order to complete the web tool we can create a web service with access to the geospatial index created. The web service would allow for easy user search via a web interface.

We will now dig into how to create a web service. This part is included here because it might be of interest being the technology used relatively new. Being the project developed in 2014, there are many more choice than the early 2000s when Php and HTML were the obligatory choice. Python is used instead together with the templating language Chameleon, CSS to manage the style, Javascript for handling client-side operations and AJAX requests. The framework of choice for the server side is Pyramid.

7.3.1 Apache, Pyramid, Django, Ruby on Rails

An overview of the competitors Pyramid and Django web frameworks is presented. In the web era where Php and HTML are fortunately not the only choice for web programming anymore, we are presented we many more alternatives for quickly setting up a web service. The landscape is vast. We go from
the old and obsolete Apache Php server which requires a more than consistent set up to the achievements reached by the Ruby On Rails framework which practically comes working and configured out of the box. When Python is the language of choice, Pyramid, Django or Flask are all reasonable alternatives. Since Flask is usually the framework of choice for micro-applications (it is referenced to as a micro-framework), and it is the least mature of the three, it will be kept out in this section.

Django among the three is the one that allows the quickest set-up. It comes with a large amount of features built-in: a templating engine, ORM set-up (it uses SQLAlchemy), a routing system, forms, database administration tools and more. It may be almost considered as plug-n-play as Ruby On Rails. Pyramid on the other end puts flexibility as its major strength, allowing developers more choice on the tools to utilize. This means that the developer can choose which database and ORM system to use, templating system, URL routing and so on. It also means that these things need to be configured though. Fortunately configuration is modular and quite fast to perform, making Pyramid a scalable, professional and flexible solution for complex applications.

Historically, Pyramid was released first in 2005, while Django came out in 2006. Despite the release order, the Django community (measured using GitHub and Stackoverflow data) is by far the larger of the two. Both frameworks are very mature and are available under BSD-like permissive licences.

For more details about python web frameworks, please see [1].

7.3.2 Python package manager and virtual environment

A brief introduction to see how python solved the packaging problem might be of great interest. Furthermore, it might be needed to understand the service set-up.

Pip

Pip is a recursive acronym that stands for “Pip install Python”. Pip is the python package manager and can be used to install and remove python modules (although external package managers such as “apt-get” might be used as well). Pip allows to specify which version of a package to install and the location from which to download them. Python packages are usually made available through PyPI (the Python Package Index). The official PyPI repositories provide thousands of free python packages. Sometimes, companies can setup their own PyPI repositories in order to have control on which packages are installed on their systems. Python versioning is introduced in the attempt of managing compatibilities between different packages. A “requirements.txt” file can be created for each Python project where to define which package and relative versions to install. Developers can therefore have fine-grained control on which versions are required for the system and upgrade packages only after successful compatibility testing. Now, in order to avoid compatibility issues, it is often necessary to use older versions of packages combined with newer versions of others. The situation can get complicated easily and python comes to the rescue with a very clever solution under the name of Virtual Environment.
Virtual environment

The Python Virtual Environment is able to create a complete environment for any python project so that dependencies do not conflict between different projects. The command line utility `virtualenv` can be used for this purpose. It should be used specifying the directory where to install the entire python packages tree. In this example, the directory’s name is “venv”:

```bash
$ virtualenv venv
```

After running `virtualenv`, a local version of `python` and `pip` are installed, and environmental variables are modified so that the local python and the local packages are used by the project. We will indicate our virtual environment as `venv` from now on.

7.3.3 Pyramid set up

As we saw in the web frameworks overview section, pyramid needs to be modularly configured. Some python packages chosen for development:

```bash
(venv) $ pip install pyramid nose deform sqlalchemy pyramid_chameleon
```

The package `pyramid` is obviously the web framework itself, `nose` is used for testing reasons, `deform` is a nice add-on used for speeding up creation of web forms. Finally we have `sqlalchemy` that provides ORM functionalities and `pyramid_chameleon` with which the Chameleon template system is included.

As a first thing, we need to write some settings for our web application. We do so by using the `Configurator` class in our configuration “__init__.py” file.

```python
from pyramid.config import Configurator

def main(global_config, **settings):
    config = Configurator(settings=settings)
    config.include('pyramid_chameleon')
    config.add_route('hello', '/hello/{name}')
    config.scan('.views')
    return config.make_wsgi_app()
```

Listing 7.2: Pyramid configuration file

As we can see in the code, pyramid is being communicated to use the chameleon template engine; to add a route called hello accessible via the URL `/hello/someString`; to scan the module “views” for adding the views where the server code should reside; to start up the wsgi server.

The pyramid server can be started by using the `pserve` terminal command that comes with the framework installation, after adequately setting up the folder structure. More details on the official documentation [24]
$pserve development.ini --reload

The `development.ini` file is only an alternative to the configuration class seen in listing 7.2.

Views

The views are parts of code (functions or methods) where the flow is routed when the user hits the correspondent URL associated to that `route`. Views are python functions or methods marked by the `@view_config` python decorator. A python decorator (in very simple words) is an operator able to take as input the decorated function and add some functionalities to it. In this case the function is marked as a view with a route associated to it and a renderer (usually a template file, but it can be a json renderer or others).

```python
from pyramid.view import view_config

@view_config(route_name='hello', renderer='hello.pt')
def hello(request):
    name = request.matchdict.get('name')
    return {'greetings': "Hello\n"+name}
```

Listing 7.3: Example view.

In listing 7.3 there is an example view. When the user hits the url `host:port/hello/someString`, the view connected to the `hello` route will be activated and the code from the `hello` view will be executed. This code will parse the request, extract the variable `name` from the request (in this case “someString”) and create a response string “Hello someString” that will be returned to the template engine. The latter is responsible for creating the actual final html that will be fed to the browser.

Templating system

The templating system are used to ease the writing of html. They allow to put some logic inside the html file, to handle variables returned by the back-end code, to include flow control and loops to write html quicker and more flexibly.

```html
<html>
<head>
    <title> DEMO</title>
</head>
<body>
```
Listing 7.4: Example template.

In the example in listing 7.4 there is a very simple example of chameleon template that renders the variable `greetings` returned by the view in the last subsection. Template engines can be rich and powerful but it is good practice to keep as much logic as possible in the back-end system for various reasons, among which better testability.
Chapter 8

Results

The following Chapter presents additional results for the benchmarks executed in Chapter 6, where a Reverse Geocoder is described. The Reverse Geocoder makes use of the Tiger database and is able to output a zipcode address given a latitude/longitude point inside the United States of America. A more advanced version might include retrieval of entire addresses.

8.1 Final benchmark comparisons

Timings for the different spatial software solutions analyzed are finally compared among each other in the following single scale charts (as opposed to complexity charts from Chapter 6). Conditions are the same as seen in the experiments in Chapter 6. For ingestion results see Figure 8.1 while for the search average results see table 8.1.

8.1.1 Bulk ingestion results

Ingestion was performed using the postal codes dataset of the United States under the same conditions (same machine, same processes running). The bulk ingestion results for the different softwares are plotted on a single-scale chart for comparison (opposed to the two-scale chart seen before). Results are shown in Figure 8.1. It is evident that PostGIS is faster than Elasticsearch in absolute value. Even though both softwares present logarithmic behaviour for ingestion, PostGIS constant factor is better.

8.1.2 Search query results

Query experiments were run using a point as input for the systems. The response time was measured over an average of 1000 tries. Experiments were run under same conditions on the same multi-core machine. The standard deviation shown in the bar chart is very small, so the results may be considered quite conclusive. Scalability under pressure and high-amount of transaction per minute was not tested unfortunately due to lack of resources. Elasticsearch is considered to be the best solution in the event of increased number of transaction due to its natural predisposition for proper scaling. For the search tests, it
is trivial to show how both Elasticsearch and PostGis solutions are way better than a linear search, see table 8.1.

This time, differently from the ingestion phase, Elasticsearch shows faster speed for query responsiveness (see Figure 8.2). This could be a determinant factor when choosing the correct software for an application whose main requirement is quickness of response. Therefore, it can maintain good scalability when accessed by a great number of users and high volume of requests.

Table 8.1: Search query times

<table>
<thead>
<tr>
<th>Query type</th>
<th>Average over 1000 query (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear search</td>
<td>9503.305</td>
</tr>
<tr>
<td>PostGis</td>
<td>16.111</td>
</tr>
<tr>
<td>Elasticsearch</td>
<td>3.854</td>
</tr>
</tbody>
</table>
8.2 Geospatial software choice

Timings for the naive approach compared to the index solutions show that it is neatly an unfeasible option 8.1. The final system of choice is Elasticsearch, for various reasons:

1. Elasticsearch provides excellent scalability and flexibility. It is so easy to boot up additional nodes in a cluster if necessary without the need of system restart.

2. Elasticsearch is provided with excellent monitoring systems. Monitoring is an important factor when considering a tool for industrial applications. Emergencies can in fact considerably impact company’s business and reputation and must be dealt with promptly. A good monitoring dashboard system can be of great help. Elasticsearch’s plugin Kibana is an eminent example of a good solution. It also allows easy ingestion of log files that can be quickly analyzed with a Graphical Interface.

3. Elasticsearch is schema-free. Again, this kind of flexibility is a great feature in industrial applications (often far different from academic ideal ones unfortunately). Requirements and specifications can change very abruptly in the landscape of the World Wide Web 2.0 and prompt adaptation can alone decide the future course of a company in the highly competitive Silicon Valley.

PostGIS is not able to deliver the numbers during search. Moreover, it is not engineered with scalability in mind. A possible alternative solution to achieve scalability and still use PostGIS as core engine might be to experiment PostGres-XL, a distributed SQL solution that is said to be able to handle high volumes of traffic.

8.3 Service choices

Conclusions and decision making process for the service implementation have extensively been discussed already in Chapter 7. Python is chosen over Ruby on Rails and Php. Pyramid is the elected Python framework of choice being it modularly configurable. We can choose the ORM system, the template system or its assets routing behaviour. Chameleon is the chosen template engine. Javascript testing can be achieved using the mocha-framework, a straightforward standard solution for Javascript applications.

8.4 Data processing achievements

In table 8.2 are some numbers about the successful results reached during the data pre-processing phases. The table shows the percentage of valid data at the beginning of the project work, and the percentage of valid data afterwards. Valid data is more than 97%.
Table 8.2: Data processing achievements. “Rough” is used because the standard deviation on the average is high.
Chapter 9

Conclusions

This document described the industrial implementation process of rich geospatial search features. Moreover, it presented a good overview of the theoretical Computer Science concepts behind geospatial data structures used in nowadays common DBMS and commercial applications. The work problem addressed was successfully tackled and solved. A perfectly functioning web-service application was the outcome of the processes described. A way to store geographical data into various database solutions was described, in addition to a method for it to be searched (the Python web-service). Data validation and quality assessment were treated.

After a background theoretical introduction in Chapter 3, a presentation of related work in Chapter 2 and a description of the design methodology in the following chapters, this last chapter presents some insights on potential future developments.

9.1 Methodology

A summary of the methodology and steps performed is presented now so to give a final overview of this document.

9.1.1 Data pre-processing

Initially, a data pre-processing phase was deemed necessary. Data ingested by partner companies was unknown and, as such, its quality needed to be assessed. This phase was called Data assessment and consisted in two different steps:

1. Data quantitative audit
2. Data qualitative audit

The quantitative audit consisted in getting detailed and complex statistics about the data itself. For example the total amount of entries presenting polygons, zipcodes, circles or any combination of them or the amount of entries for which the geospatial search feature would have to be enabled in the end. All these results are counting something and are therefore part of the quantitative audit.
The qualitative audit consisted in getting an idea of the quality of the various entries in the database. For example it was useful to know how many polygons were broken or how many entries presented void geographic feature. Moreover building a visualization tool was also part of the qualitative audit. It was in fact useful to get an idea of the shape of the geo-data.

9.1.2 Data validation

After getting detailed statistics over the quality of the data, measures were taken in order to improve the percentage of valid data so to be able to use the highest amount of entries in production. New heuristics are presented in this thesis. These heuristics were inserted into the Data Ingestion company code so to improve data validation during ingestion time.

9.1.3 Related work study

All the decisions about how to proceed were taken after careful related work study. This is a wide-used and well-known process in modern agile companies and the Internet network is a necessary tool for this step. Some of them are presented in this document. Existing comparisons between tools are sometimes the right method to proceed. Sometimes these comparisons were not to be found. Hence experiments to compare various solutions were conducted.

9.1.4 Experiments

A few experiments were conducted in order to choose the right tool for implementing geospatial search. The competing solutions were Elasticsearch and PostGIS (extension for Postgres). The experiments results are presented in the Results Chapter 8. The experiments outcome brought to the choice of Elasticsearch over PostGIS.

9.1.5 Web service implementation

A web service was implemented in order to put in one place monitoring tools for the Data assessment and Data validation phases. Moreover the service included the visualization tool and a prototype of the final search tool useful for debugging purposes. The web-service solution was chosen in order to satisfy the goals of share-ability and accessibility requested. All developers could then easily interact with it.

9.2 Future Work

Some insights on possible future developments are given, focusing on improving the Geocoder application and improving geospatial features in DBMS solutions. Finally, a few ideas are suggested about possible data mining features for geospatial data.
9.2.1 Reverse Geocoder future complex street address version

As a first thing, the Reverse Geocoder might benefit of improvements. As for now in fact, it is only supporting zipcodes, given input coordinates. A true reverse geocoding should be able to return complete addresses instead. The implementation should be trivial. First, we could obtain the entire datasets for streets, civic numbers, cities, regions. Then, we could build geospatial indexes over each one of them as seen previously. Finally, a quick search in each of the datasets would hand out the result. This is one of the many way that we could proceed on. Of course there are many alternatives instead of building a reverse geocoder from scratch. First of all, many companies like Bing or Google offer this kind of services out of the box through convenient APIs. The application might be more robust if relying on pre-existing, well-tested APIs but the downside is that money must be paid for these services.

9.2.2 Geospatial features improvements in DBMS

Geospatial noSQL tools are still very very young as of 2014. There is large space for improvement and future work. To make a simple example, a basic bug in the implementation of the math for simple geo operations was discovered in Elasticsearch 1.2 version. The bug consisted in a wrong factor 2 while creating objects of type Circle. Due to the buggy factor 2, the “radius” parameter was treated as a diameter. Fortunately the bug was fixed in successive Elasticsearch releases but it is a clear index of the youth of this system when used for geospatial applications. It would be very interesting to see its performance benchmarked more precisely in a benchmark-dedicated dissertation. It would be very very interesting to provide an easy guide to spatial solutions choice. Unfortunately it does not seem to exist the perfect guide towards the perfect geo-tool based on the project requirements. As an example it would be nice to have some prospect outlining the best tool when high-volume dynamic insertions are a requirement. Or when fast search is the most needed feature. Or again when the data is highly-dynamic and quickly changing. In this dissertation, while implementing rich spatial search features, some of those questions are answered. Still, a work mainly focused on guiding towards a better informed choice might probably be of great use in the field.

9.2.3 Advanced geospatial features in DBMS

Another direction research could proceed towards is the development and inclusion of advanced geospatial features in commercial DBMS actually in use today so to ease developer’s work. In fact, the spatial operations and queries seen previously in this work are only a few of the many possible ones that were historically studied (they might be integrated). Just as an example, we recall all the operations nowadays contemplated by research studies, such as Distance Operators, Directional Operators and all the Topological (for a complete list see Chapter 2).
9.2.4 Geospatial Data Mining

In the literature study phase of the work, papers and previous works treating of how to mine spatial data were sought. It turned out that the literature on the subject seems to be very poor, being spatial data difficult to mine and being the field quite young still. All found papers only present slightly different varieties of spatial data visualization in order to find patterns, often lightly combined with classic data mining techniques. Unfortunately classic data mining algorithms have been proven quite inefficient applied to spatial data. Therefore, here a different approach is tried, bringing to the building of the visualization tool that can help humans to manually find various data patterns [8]. As for future work, the possibilities for research are innumerable. Spatial data is implicitly present in most of the phone applications that users interact with every day. Top hi-tech companies like Google, Apple, Facebook have understood the enormous potential of collecting geographic information and have started to make use of it (see the Google Knowledge Graph [13]). There is no doubt that the Big (Geo) Data contains a whole world of hidden information, currently inaccessible. It would be of great help directing efforts towards solving the geospatial data mining problem.
Bibliography


