Bachelor in Telematics Engineering
2014-1015

Bachelor Thesis

“Predictive analysis for cache generation in quadtree based geo visualizations”

Author:
Carla Iriberri Gutiérrez

Supervisor:
Víctor Elvira Arregui

“Visualization leads to comprehension.”

To my parents:
for your efforts.

Abstract

The Web Mercator projection is commonly used in geo data visualizations. This projection shows geospatial information based on some restrictions, but allows to scale the whole world with a precision of centimeters. One of the most important restrictions is the clustering of the information within cells inside map tiles, so that they can be cached. Depending on the data that is being visualized, that cache must behave in a different way to be responsive to user requests.

Web mapping has become a very relevant market in the last few years, and it is expected to grow even more in the near future, with a great variety of applications such as self-driving cars or business analytics. The analysis of user reactions and interactions against a map filled with information will be key when building the mapping platforms of the future.

This project aims to predict usage patterns in web map visualizations so that dynamically generated data can be precached to improve the usability of such maps. To that end, we use different statistical approaches, all of them trained with real data of map usage activity.

Throughout this document, we describe a suitable way to curate the raw data acquired through thousands of log files and how to obtain statistical insights from it. Based on this data, we define a predictive model for tile caching, and we test it against real map usage data. We also cover the timing details of tile generation, comparing the efficiency of serving cached tiles against generating tile images on runtime. The economic impact of the developed solution is also explained in terms of how much it costs to store the tiles predicted by our caching algorithm in a content delivery network. Regarding the social context, we analyse which legal frameworks regulate the web mapping industry and how do them affect our caching solution.

1. Motivations and goals

The main objective of this project is to build an algorithm to predict which map tiles need to be generated and cached whenever a web map is created. In order to achieve this, we will analyse the spatial data contained on thousands of maps together with the interactions that the users perform on them: zooming, moving or clicking on the map features.

1.1. Problem statement

Predicting which tiles are likely to be requested when a map is generated will allow caching these tile resources in advance and avoid their generation on runtime as a consequence of a user request. The response time of a tile that is cached versus the response time of a tile that needs to be generated could be different in several orders of magnitude depending on the complexity of the data that needs to be rendered in a tile.

This research on possible enhancements that could be applied to the current web mapping platforms and tools gets its motivation from an industry field that is expected to grow within the next years.

2. Introduction to web mapping

2.1. State of the art

Cartography has been a part of human history for a long time, the earliest known map of Earth being dated in the year 6200 B.C. From these ancient maps of Babylon, Greece or Asia and on into the current century, the different societies that have populated the planet have created and used maps as tools to describe their world.

The improvements in technology and the advent of the Internet have provided a new way to communicate and work which has affected the manner in which maps are built nowadays.

In 1996, a company named MapQuest released the first web map, which provided also geocoding and routing capabilities. This map was created by a set of images that were explorable through direction arrows located in each of the four borders of a squared screen. Browsing around this map required a full site refresh due to the way that the complete images per each region were requested to the servers.

During the following 10 years, other companies such as the NASA or ESRI released their own web map services. In 2004, the open source world map project OpenStreetMap was founded by Steve Coast, allowing the people in the world to collaborate and map their local data. Just one year later, in 2005, the first version of Google Maps was released. Google Maps was introducing therefore the ideas that changed the web mapping world and that have become an standard these days.

Along with the new Google Maps service, Google presented the concept of “tile”. Instead of drawing the maps by showing a single image, their service used tiles: small squared images that placed side-by-side made up the complete map.

This scalable approach allows loading small predefined images instead of images of millions of pixels which would be too large to download or hold in memory at once. The specifics about how the tiles were defined constitute what is known as “Google Maps tile convention”:

- Tiles are images of 256x256 pixels
- Tiles are drawn according to the Web Mercator projection
- Tiles are ordered hierarchically according to the zoom level applied to the map

Together with these statements, other tiled web map standards adapted more specific conventions:

- Tiles are presented as PNG images
- Tiles are described by an ZXY numbering scheme, where Z is the zoom level and X and Y identify the tile

Tiles are organised in a quadtree scheme. A quadtree is *"a tree data structure in which each internal node has exactly four children"*[1]. This hierarchical structure implies that at each level of zoom a web map will be formed by 2^{2z} tiles. At the level of zoom zero, the whole world can be drawn just with a single tile ($2^0 = 1$). Four tiles are necessary to build the world at the second level of zoom. The amount of tiles needed to draw the world when the zoom level increases grows exponentially.

Each tile can be always identified and requested by means of its (Z, X, Y) triplet. This system of coordinates defines the different set of tiles that make up the whole world for each one of the levels of zoom available (usually, the range of zoom goes from 0 up to 24). The following figure demonstrates visually the organisation described above:

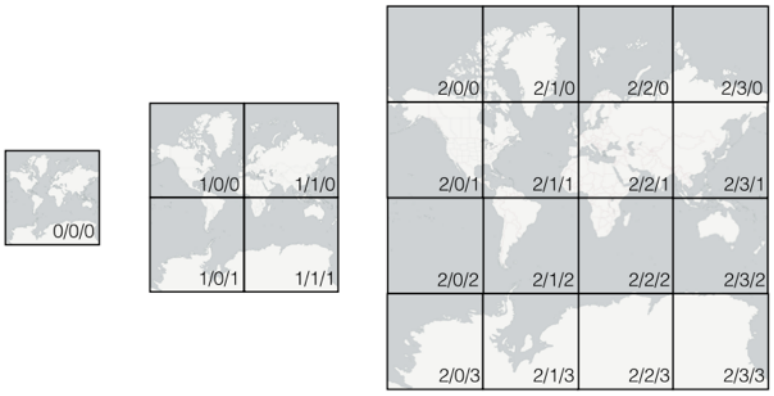


Figure 2.a): Schema: Quadtree based hierarchy for tiles.

The fact that these tiles have specific boundaries for the geographical regions makes the system standard against different tile providers. Tiles are generated in tiler servers and offered through a REST API with URLs that include the three ZXY parameters. An example of these URLs can be http://tile.openstreetmap.org/4/2/6.png, which corresponds to a tile in the zoom level 4 whose position is (2,6).

Analogously, the rest of tiles of this service can be obtained through the parametrised URL http://tile.openstreetmap.org/{z}/{x}/{y}.png.

A web map is formed by one or more layers of tiles which can be of two different types. Base layers are intended to show the geographical information of the world. Data layers show data on top of a base layer.

The common layer that most of web maps share is the base map (or base layer). Base maps are built by static tiles that are usually rendered once and then stored in a cache in order to serve them when necessary. Base maps can be obtained from satellital imagery or designed using the data available throughout the world as a source. Companies such as Stamen or Mapbox design base maps and create software (like *Tilemill*) to allow people to customise their own base maps by using OpenStreetMap open data. The next figure shows different base maps for the same geographical boundary (the same tile) in the area of the San Francisco Bay.

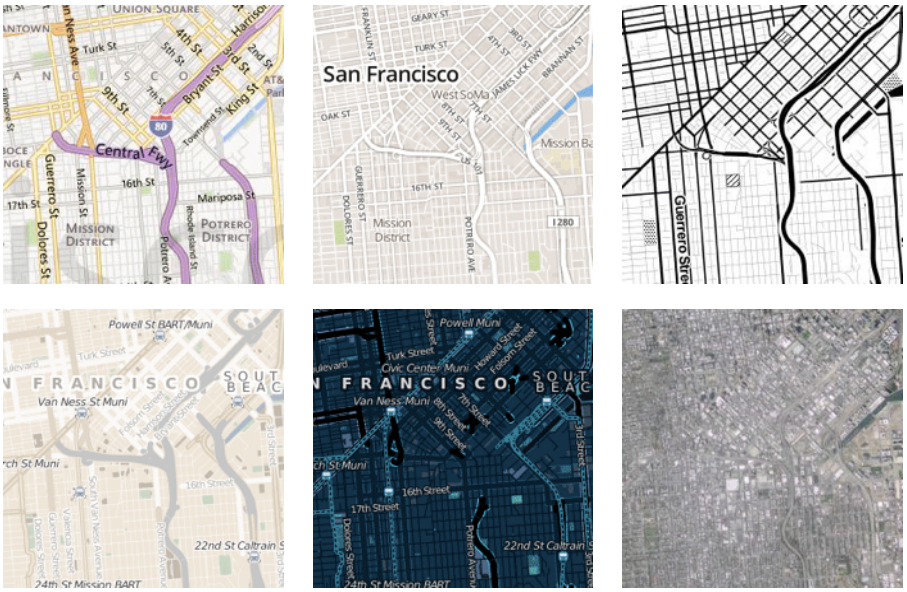


Figure 2.b): Example of different designs for the same tile.

As web maps are used to visualise information, we can find more tile layers on top of the base maps. The base layers help us to understand the data and analyse it with a geographical meaning.

There are several companies in the market whose product is based on displaying data tiles generated from user data. With this objective, companies like ArcGIS, Mapbox or CartoDB serve millions of tiles per day[2].

Companies such as Twitter, Facebook, Uber or Tesla are making a great use of maps: from creating graphs that show where "hashtags" are used over time[3] to where the taxi drivers are picking up their clients and driving them up to their destination. Thanks to the global connectivity and the use of location services, human beings have become sensors: all this collected data is associated to a geographical meaning that is able to offer powerful insights. Mapmaking has unshackled itself from the confines of paper atlases and migrated onto cellphones, which are, after all, GPS units that also make phone calls.

The web mapping industry is evolving jointly with the car industry, whose near future challenge is focused in self-driving cars: collecting and analysing data in order to build intelligent routing services will be the key to find driverless vehicles in our streets, and it will be supported by the mapping companies in the market.

Maps form the heart of consumer businesses like Uber and Airbnb. They are central to aerospace and defense firms. Cartographers are becoming integral in fields such as media, finance and health. All that attention is expected to drive U.S. revenue in the geospatial industry to nearly $100 billion by 2017, according to a 2012 report from the Boston Consulting Group commissioned by Google[4].

In this scenario, being able to know which tiles will be requested taking into account different parameters such as the characteristics of the underlying data, the kind of represented geometries or their spatial distribution could be the key to build scalable and fast platforms to the map services of the future. Throughout this project we will process, analyse and define statistical approaches for samples of real datasets in order to discover the way to predict the tiles that would be requested taking into consideration the characteristics of a given map.

2.2. Technical overview of web mapping

In this section we will explain the basic concepts of web mapping: from the data formats used to build a map, up to the storage of the generated tiles in the latest generation of content delivery networks (CDNs).

2.2.1. Geospatial vector data

Vector geometries are used in GIS in order to represent the spatial component of geographical features, such as lines, points or polygons. The Open Geospatial Consortium[5] standardised[6] how vector geometries are expressed.

In a human readable way, a geometry can be represented in the *"Well-known text"* markup (WKT). A well-known text geometry string includes the type of the represented geometry and a collection of its vertices (or, in case of a single point, the point itself). The vertices are written as coordinates corresponding to a coordinate reference system which must be specified (such as the Universal Transverse Mercator coordinate system — UTM).

Geometry type	Text literal representation	Description
Point	POINT (10 10)	A Point
LineString	LINESTRING (10 10, 20 20, 30 40)	A LineString with 3 points (or vertices)
MultiLineString	MULTILINESTRING ((15 15, 25 25), (0 0, 10 15))	A MultiLineString with 2 LineStrings
Polygon	POLYGON ((10 10, 10 20, 20 20, 20 15, 10 10))	A Polygon
Multipoint	MULTIPOINT ((10 10), (20 20))	A MultiPoint with 2 points
MultiPolygon	MULTIPOLYGON (((10 10, 10 20, 20 20, 20 15, 10 10)), ((60 60, 70 70, 80 60, 60 60)))	A MultiPolygon with 2 polygons

Table 1: WKT geometry examples

Geometries can also be expressed as a stream of bytes, which allows to use them in geospatial applications in binary form. This standard representation is named *"Well-known binary"* (WKB) and the geometries expressed this way are named

WKBGeometries. This sequence of bytes in hexadecimal form follows a specific format: The first byte of the sequence indicates the order of the most significant byte with an integer: big endian (00) or little endian (01) if the most significant byte comes first or if it is the latest one, respectively.

The following 4 bytes in the WKB standard are reserved for the geometry type, expressed as an uint32. The WKB integer codes for the geometries we presented in *Table 1* are defined as follows:

Geometry type	WKB integer code
Point	1
LineString	2
MultiLineString	5
Polygon	3
Multipoint	4
MultiPolygon	6

Table 2: WKB integer codes

After these first 5 bytes, which are fixed for all geometry types, the next ones depend on the geometry that needs to be described, as each of them has a different data structure. The most fundamental type, the point, forms the basis for all the other types. While a *Point* can be defined with two double numbers (8 bytes each) for x and y coordinates, a *LineString* is defined by its number of points (uint32, 4 bytes) and an array of *Point* elements.

The following graph represents the byte representation of a single *Point*:

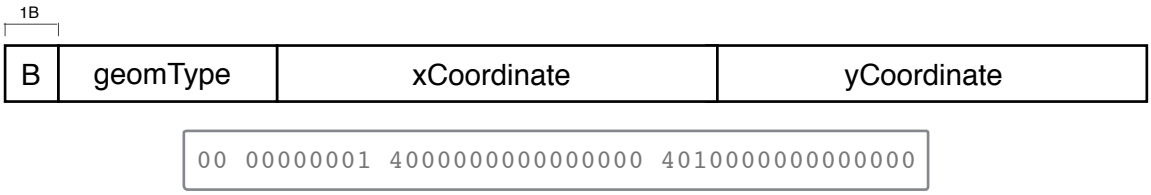


Figure 2. c): WKB structure of a point in coordinates (2, 4)

Further details of the specific data structures for *WKBPoints* and *WKBLineString* are shown in the following figure:

```
Point {
 double x;
 double y
}

WKBPoint {
   byte byteOrder;
   static uint32 wkbType = 1;
   Point point
}

WKBLineString {
   byte byteOrder;
   static uint32 wkbType = 2;
   uint32 numPoints;
   Point points[numPoints]
}
```

Figure 2. d): WKB data structures

As we have studied above, storing geometries in a database can require a lot of space as all the coordinates are being saved as doubles (8B). With the objective of avoiding this waste of space and improving the redundant specification of coordinate information, a group of individuals is working since 2013 on the definition of a new format to define geometries as a byte sequence. The Tiny Well-Known Binary [7] (or TWKB) format has as its main principle to store the absolute position of a point once and refer to the rest of the points in the same geometry as delta values relative to the initial point.

2.2.2. Geospatial data formats

In the traditional world of geographical information systems, the most common format for geospatial data is the Shapefile. GIS has been using this format since it was introduced by ESRI in the early 1990s. Web maps prefer the KML format, or more recently, GeoJSON.

These most common formats in web mapping have their own characteristics which are defined as follows:

- **Shapefiles**

The Shapefile[8] format is a multi-file format — it consists of a set of files with the same name and stored in the same directory which are differentiated by their extension.

A Shapefile has to be formed, at least, by a *.shp* file, a *.shx* file, and a *.dbf* file. These files contain the geometry data, the indexes and the attributes, respectively. Other auxiliary files, as the *.prj* one, are not mandatory and contain extra information for the Shapefile, as in this case, its spatial projection.

- **Keyhole Markup Language**

The KML (Keyhole Markup Language)[9] format relies on the XML notation and adds to it a geographical meaning by being able to define *features* such as points, polygons, lines or 3D models. It was originated in the Keyhole company to use it in their Keyhole Earth Viewer 3D application. The company was acquired by Google in 2004 and the program was renamed to Google Earth. From 2008, the KML format became an standard of the Open Geospatial Consortium.

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
<Document>
<Placemark>
  <name>New York City</name>
  <description>New York City</description>
  <Point>
    <coordinates>-74.006393,40.714172,0</coordinates>
  </Point>
</Placemark>
</Document>
</kml>
```

Figure 2. 3): KML sample file

- **GeoJSON**

The GeoJSON[10] format is an open standard geospatial data interchange format based on JavaScript Object Notation (JSON[11]) that allows encoding geographical features and their metadata. The GeoJSON format working group was originated in March 2007 and the format specification was complete in June 2008.

```
{
    "type": "Feature",
    "properties": {
        "name": "Coors Field”
  },
    "geometry": {
        "type": "Point",
        "coordinates": [-104.99404, 39.75621]
    }
}
```

Figure 2. f): GeoJSON sample file

Besides these formats, maps can be created from a variety of other sources: Comma Separated Value (CSV) files, simple JSONs, or Excel spreadsheets (which are converted to CSV files) are commonly used as data sources that can be geocoded — translated to a point, a line or a polygon with specific coordinates — if they contain geospatial information like city names, postal code, street addresses, administrative regions or even IP addresses.

2.2.3. Storing and querying geospatial data: PostgreSQL and PostGIS

In GIS, as well as in almost any other discipline, relational databases are the most used type of database[12]. Working with geospatial information that needs to be stored implies the challenge of working with the aforementioned geometry types that were mentioned in a database context. There are several options to fulfil this need: PostGIS stands out in the open source world, while Oracle Spatial offers a private solution. Other databases as MySQL do not accomplish the required *Simple Features Access* standard[13] of the OGC and cannot be considered as a complete spatial database manager but it is often used as a geographic information container.

In this section we will highlight PostGIS[14], which is the database over which the CartoDB platform is built. PostGIS is a module for the open source PostgreSQL database. This module provides PostgreSQL the means to store geometries in the database and the ability to perform geographical analysis computations. Its first stable release took place in 2005 but it is in constant development.

There is a large number of applications that use PostGIS in the backend, such as ArcGIS, CartoDB, QGIS or Tilemill, developed by the Mapbox company, which

include both open source and proprietary software on both server and desktop systems[15].

PostgreSQL supports natively geometric types as well as functions and operators. In the database, the geometries are defined as shown in the following table[16]:

Name	Storage size	Representation	Description
Point	16 bytes	Point on the plane	(x,y)
Lseg	32 bytes	Finite line segment	((x1, y1), (x2,y2))
Polygon	40+16n bytes	Polygon (similar to closed path)	((x1, y1),…)
Line	32 bytes	Infinite line (not fully implemented)	((x1, y1), (x2,y2))
Box	32 bytes	Rectangular box	((x1, y1), (x2,y2))
Path	16+16n bytes	Closed path (similar to polygon)	((x1, y1), (x2,y2))
Path	16+16n bytes	Open path	[(x1, y1), (x2,y2)]
Circle	24 bytes	Circle	<(x,y),r> (center and radius)

Table 3: PostgreSQL geometry types

The set of functions and operators available in PostgreSQL can be used to perform various geometric operations such as scaling, translation, and determining intersections. The most common ones are shown in the following table[17]:

Operator	Description	Example
<->	Distance between	circle '((0,0),1)' <-> circle '((5,0),1)'
*	Scaling/rotation	box '((0,0),(1,1))' * point '(2.0,0)'
#	Number of points in path or polygon	# '((1,0),(0,1),(-1,0))'
@@	Center	@@ circle '((0,0),10)'
&&	Overlaps?	box '((0,0),(1,1))' && box '((0,0),(2,2))'
##	Closest point to 1st operand on 2nd operand	point '(0,0)' ## lseg '((2,0),(0,2))'

Table 4: PostgreSQL common geometric functions

2.2.4. Styling geometries: CartoCSS

Once we have analysed how geometries are defined and stored, we will study how geometries can be drawn. In the following section *Generating map tiles* we will understand the internals on how maps and tiles are generated with the Mapnik[18] renderer, but this section will be focused in geometry styling.

CartoCSS[19], also known as *the Carto language*, is a language for map design which is similar in syntax to CSS but builds upon it with specific abilities to filter map data. It was created by Tom MacWright, Konstanting Käfer, A.J. Ashton and Dane Springmeyer, which are all members of the Mapbox company team.

In a similar way as CSS is used over DOM elements, CartoCSS is used over layers. Different properties are available for different types of geometries, such as points, lines and polygons, which have specific attributes.

CartoCSS has also the capability to define variables, nest styles or filter the geometries depending on their data attributes. In the following images we will show how different CartoCSS code creates geometries with different styles:

```
#barcelona_building_footprints{
  polygon-fill: #FFFFFF;
  polygon-opacity: 0.7;
  line-color: #000000;
  line-width: 1;
  line-opacity: 1;
}
```

Figure 2.g): Black/white map and CartoCSS code

```
#barcelona_building_footprints{
  polygon-fill: #5CA2D1;
  polygon-opacity: 0.7;
  line-color: #081B47;
  line-width: 1;
  line-opacity: 1;
}
```

Figure 2.h): Blue map and CartoCSS code

This language allows drawing the different types of geometries by using their own properties. In the examples above, we can see the usage of the most basic polygon and line characteristics, but points can also be styled with properties like *marker-width*, *marker-fill*, or *marker-opacity*.

Notice that CartoCSS properties are written inside the scope of the layer that is going to be styled. These scopes are not only referred to specific layers, but they can also be used to filter the data of the geometries. These selectors that allow to filter the data can understand specific values and comparisons (like *[area > 10]* or *[name = 'Barcelona']*) but they also support regular expressions. For example, filtering data by the length of its *name* attribute could be achieved by using a selector like *[name=~'^.{10, }$']*.

Another use of CartoCSS selectors which is very common in a map is to apply constraints depending on the level of zoom. This way, the properties specified within the scope *[zoom = 4]* will only be applied when this equality holds.

As an example of these defined selectors, the following figure shows a map colorised by the area of the polygons (which correspond to the buildings in Barcelona):

```
#barcelona_building_footprints{
  polygon-fill: #FFFFB2;
  polygon-opacity: 0.8;
  line-color: #FFF;
  line-width: 0.4;
  line-opacity: 1;
}
#barcelona_building_footprints [ area <= 1152701.46653] {
   polygon-fill: #B10026;
}
#barcelona_building_footprints [ area <= 799.513571955] {
   polygon-fill: #E31A1C;
}
#barcelona_building_footprints [ area <= 435.76472095] {
   polygon-fill: #FC4E2A;
}
#barcelona_building_footprints [ area <= 305.993911112] {
   polygon-fill: #FD8D3C;
}
#barcelona_building_footprints [ area <= 216.382241189] {
   polygon-fill: #FEB24C;
}
#barcelona_building_footprints [ area <= 155.74910497] {
   polygon-fill: #FED976;
}
#barcelona_building_footprints [ area <= 103.956183639] {
   polygon-fill: #FFFFB2;
}
```

Figure 2.i): Building areas in Barcelona and CartoCSS code

2.2.5. Generating map tiles

Once we have studied how the data is presented and how its styles can be defined, we will move on to the specifics of tile generation.

Mapnik is an open source map rendering engine used to make tile sets that supports various types of data files and spatial databases. It is an essential part of the stack of the different web mapping solutions that we have already mentioned in

previous sections, like OpenStreetMap, Mapbox, CartoDB, Stamen, MapQuest or Kosmtik.

In terms of styling, Mapnik is not capable of generating tiles directly from CartoCSS but from XML files, known as Mapnik XML. A Mapnik XML file will have a root node *Map*, parent of two types of nodes: *Layer* and *Style*. The *Map* node contains basic information as the background color of the map and its spatial projection. It can have one or more *Layer* children, and one or more *Style* children.

The *Style* nodes are the ones equivalent to CartoCSS. They are formed by *rules*, and can also include *filters*. In the following figure a sample of a *Style* node is shown:

```
<Style name="highway-directions">
    <Rule>
      <Filter>[oneway] = 'yes' or [oneway] = 'true' or [oneway] = '1'</Filter>
      <MaxScaleDenominator>10000</MaxScaleDenominator>
      <LineSymbolizer>
        <CssParameter name="stroke">#6c70d5</CssParameter>
        <CssParameter name="stroke-width">1</CssParameter>
        <CssParameter name="stroke-linejoin">bevel</CssParameter>
        <CssParameter name="stroke-dasharray">0,12,10,152</CssParameter>
      </LineSymbolizer>
    </Rule>
 </Style>
```

Figure 2. j): Mapnik XML style example

The *Layer* nodes specify where the data can be obtained from, for example, accessing to a PostGIS database and running a query on a certain table. A *Layer* node is presented in the following figure:

```
<Layer name="water_areas" status="on" srs="+proj=merc +a=6378137 +b=6378137
+lat_ts=0.0 +lon_0=0.0 +x_0=0.0 +y_0=0 +k=1.0 +units=m +nadgrids=@null
+no_defs +over">
    <StyleName>water_areas</StyleName>
    <Datasource>
      <Parameter name="type">postgis</Parameter>
      <Parameter name="password">****</Parameter>
      <Parameter name="host">localhost</Parameter>
      <Parameter name="port">5432</Parameter>
      <Parameter name="user">username</Parameter>
      <Parameter name="dbname">gis</Parameter>
     <Parameter name="table">(select * from planet_osm_polygon order by
     z_order) as water_areas</Parameter>
    </Datasource>
  </Layer>
```

Figure 2. k): Mapnik XML layer example

The following figure shows in a simple way the inputs and outputs of Mapnik: it obtains data and styles, and from these, tile images are rendered:

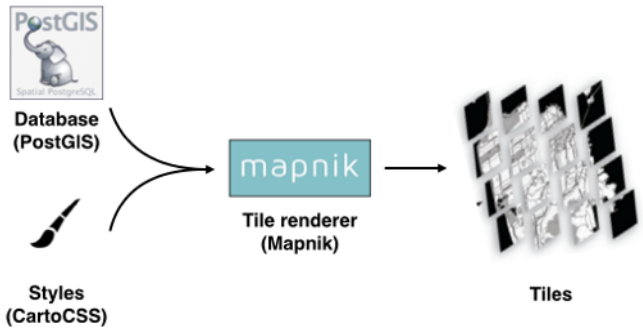


Figure 2. l): Tile generation simple schema

The CartoDB stack that we are using as example in this approach to web mapping uses an intermediate layer above Mapnik: Windshaft[20].

Windshaft is Node.js map tile server for PostGIS with CartoCSS styling. In order to generate a map, it uses a configuration file just like the XML file of Mapnik, containing the information of the data layers that are going to be rendered and their styles. These configuration files are then internally translated to the Mapnik language, which finally renders the tiles.

The specification of the layers of a map in these configuration files determines what we will call from now on a *layergroup*. Therefore, the word layergroup references the layers that will be rendered in a tile: both the SQL from which the data is obtained and the CartoCSS code.

An example of a layergroup can be seen here:

```
{
   "layers": [{
            "type": "cartodb",
            "options": {
                "cartocss_version": "2.1.1",
                "cartocss": "#layer { polygon-fill: #FFF; }",
                "sql": "select * from european_countries_e"
            }
        }]
}
```

Figure 2. m): Layergroup configuration example

Once sent to the tile server, this configuration will be saved with an alphanumerical identifier like "c01a54877c62831bb51720263f91fb33:0". This identifier is named *layergroup ID*, and it is the key to building the URL of the tiles in the system.

This way, the tiles of the map that has been configured will be accessed through a specific URL that will be associated to the styles and data of the map through this identifier. This relationship will allow us to study and analyse the requests obtained in the logs by translating them to the tile information from which the map has been created.

A sample URL could be as follows: *http://username.cartodb.com/api/v1/map/c01a54877c62831bb51720263f91fb33:0/{z}/{x}/{y}.png*.

2.2.6. Interacting with map tiles

Besides from tile rendering, Windshaft and Mapnik also take care of the interactivity of the map. When a user clicks or hovers a web map, usually the underlying information is shown as an interactive text box in the website or as a popup in which the information for a specific geometry is included.

In tiled maps, interactivity is obtained from the UTFGrid[21]. The UTFGrid is an invisible tile layer made up of arbitrary letters which are indexes into the clickable data of a map. Its specification was created by the Mapbox team with the idea of generating ASCII tiled layers for slower browsers and machines which can not cope with rendering the polygons used to draw vectors on the map tile.

The UTFGrid uses a grid-based approach where the information for each pixel is stored and uses JSON as a container format. It is exclusively geared towards the standard tiles of 256x256 pixels.

To associate data with each pixel, each feature is associated in a grid tile with a character, usually starting with 0, and then a mapping of that character is included to the actual data[22]. When each pixel in the grid is associated to a symbol, a UTFGrid layer looks as shown in the following figure, where we can see Norway,

Sweden, Denmark and the mid regions of Europe, each of them defined with a different character. Norway is drawn with the character "!", Sweden with "#" and France with the number 2, among other countries.

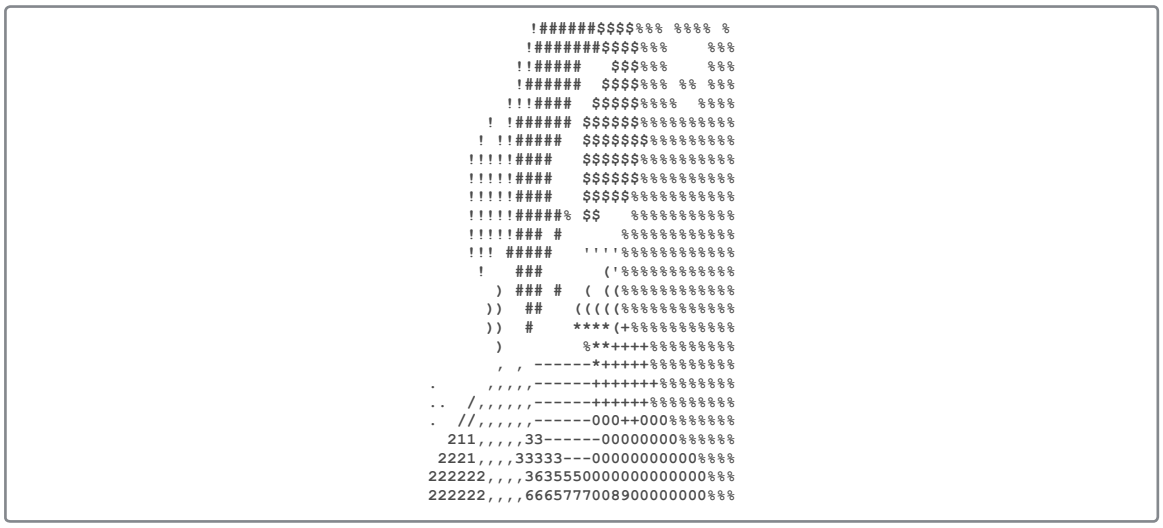


Figure 2. n): ASCII UTFGrid

Each one of those characters corresponds to a number that is afterwards mapped to a key, usually the identifier of the geometry from which we can retrieve the rest of data when any interaction occurs with its pixels. In terms of the UTFGrid object, this corresponds to "keys" and "data":

```
keys: ["9","10"],
data: {9: {cartodb_id: 48},10: {cartodb_id: 23}]
```

Figure 2. o): Keys and data mapping

The requests to this UTFGrid from the user's browser will allow us to analyse the behaviour of a user with a specific map: which regions were hovered or which ones were clicked on. Together with the requests for tiles of different positions and zooms, we will be able to reproduce and study the user's activity over a map.

Figure 2. p): Interactive map example: Nuclear reactors operating around the world

2.2.7. Javascript mapping libraries

After learning how tile sets are created, the next step revolves around how the tiles and the different interactivity options are put together to create interactive web maps.

Leaflet[23] is the leading open-source JavaScript library for interactive maps. It is developed mainly by Vladimir Agafonkin, among different contributors, since more than 5 years ago.

The creation of a map in Leaflet can be done with a couple of lines of Javascript. The following figure demonstrates how to create a map object, assign an OpenStreetMap basemap to it and add a marker in some specific coordinates:

```
var map = L.map('map').setView([51.505, -0.09], 13);

L.tileLayer('http://{s}.tile.osm.org/{z}/{x}/{y}.png', {
    attribution: '&copy; <a href="http://osm.org/copyright">OpenStreetMap</a>
contributors'
}).addTo(map);

L.marker([51.5, -0.09]).addTo(map);
```

Figure 2.q): Leaflet map creation example

The open source CartoDB.js[24] and Mapbox.js[25] libraries are built on top of Leaflet — they provide all the mapping functionality available in the Leaflet library plus specific functions that make easier to work with the maps created in the CartoDB or Mapbox platforms, respectively.

These libraries work asynchronously: the tiles and the interactivity information is requested with AJAX calls when required after a user interaction: if the user zooms in, the tiles for the next level of zoom are requested and placed in the correct placement in order to build the map.

These requests of tiles and UTFGrid data will be analysed in the following section with the objective of improving the performance of web maps through the study of user interactions.

2.2.8. Caching map tiles

Web access times, and web map access times in specific can be slow because of different reasons: the time that is taken in the generation of the requested resource (always considering that the resource is not static), transmission delays due to congested or slow links, and overloaded servers.

One approach which is commonly used to lessen this problem is to replicate the content expected to be requested in different servers. Content delivery networks (or CDNs) are considered an efficient schema for content distribution and are based in a network of distributed servers throughout the Internet in which data is replicated[26].

CDNs provide updated data by refreshing the resources of the servers in their network whenever any of them is out of date. This refreshing process — taking place in content delivery networks or in any other caching schema — is usually referred to as "invalidation", and allows old or outdated resources to be marked as no longer valid for them to be provided to the users. CDNs are location aware and the data is always provided to the end user by the nearest available server in order to minimize transmission delays.

Caching dynamic resources allows the server of a platform to skip generating these requested resources at runtime, as they get cached after the first generation and the cached data is provided until it is invalidated by an edition or deletion on its source. In this scenario, caching reduces the cost of rendering map tiles at its minimum, where only one copy of the data needs to be generated regardless of how many requests it gets.

Map tiles are defined by their layergroup and timestamp, which we have studied in the previous section. These two values will allow a cache (or a CDN) to identify a resource and to be able to determine if it contains valid or outdated data to be served.

The network schema below (*fig. 2. r*) demonstrates the following workflow:

- User A, User B, Windshaft (tiler) and the CDN are the main stakeholders, while Varnish is an internal cache of the platform that stores data that flows through the network level.
- (1) User B requests a map (a specific layergroup)
- (2 and 3) The tiler (*Windshaft*) checks the last updated time for the data
- (4) The tiler returns the valid layergroup ID and timestamp
- (5) User B requests the tiles with the temporal identifier
- (6) The tiler responds back with the generated tiles. Along the path that this request follows, tiles are cached at the different platform and network levels

After some time:

- (7) User A requests the same map
- (8 and 9) The tiler checks the last update timestamp for the data
- (10) The tiler returns the valid layergroup ID and timestamp
- (11) User A requests the tiles with the temporal identifier
- (12) The CDN intercepts this requests and responds back with the cached tiles, as the timestamp proves the validity of the data

Afterwards:

- (13) User B, which is the owner of the map, updates the data from which it is generated

Then:

- (14) User A accesses again to the map
- (15 and 16) The tiler checks the last update timestamp for the data
- (17) In this case, the data has been edited, so a different timestamp than the one obtained previously is sent back to the user
- (18) User A requests the tiles, by using the updated information

While the graph does not manifest the following steps, similarly to the response in (6), the tiler would generate the new tiles from the updated data and would send them back to User A. Notice the cache status information shown at each hop, where a white circle applies for a cache miss and a black circle does it for a cache hit.

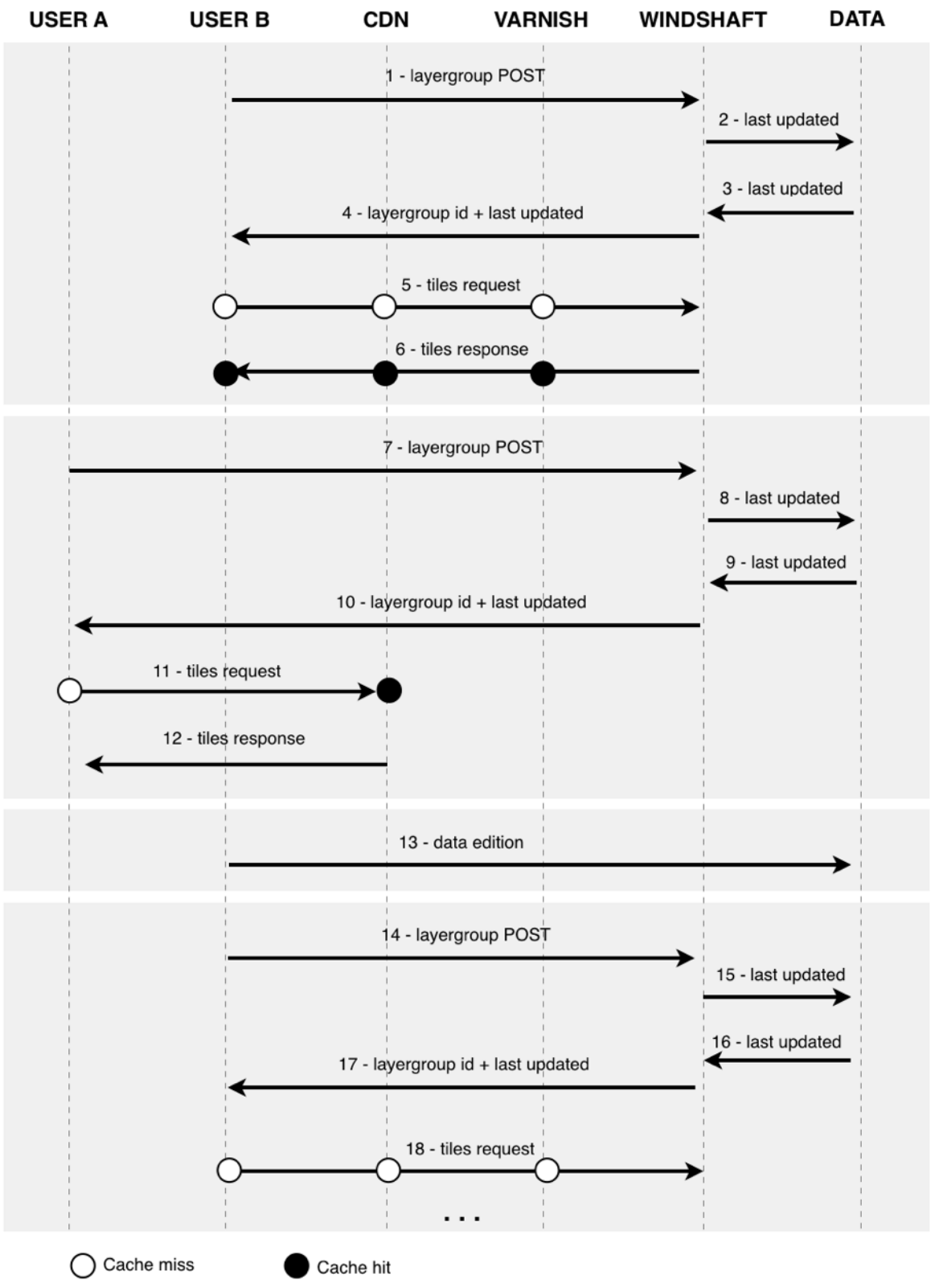


Figure 2. r): Tile cache request schema

2.3. The social environment of the web mapping market: regulatory framework

Our tile prediction solution and the web mapping market in general is not affected directly by any regulatory framework involving the work of companies or individuals.

Nevertheless, there is a Spanish law and a European directive that regulate how local governments offer Open Data and web mapping services. This Spanish law is known as *LISIGE*[27] (*Ley sobre las infraestructuras y los servicios de información geográfica en España)* and regulates the national cartographic system. It requires public administrations to publish their cartography by means of standard web services and by following preset conditions.

In Europe, the *INSPIRE*[28] (*Infrastructure for Spatial Information in the European Community)* directive was introduced in May 2007. It establishes an infrastructure for spatial information in Europe in specific areas such as metadata, data specifications, network services or data and service sharing. At the moment, it is being operated by the 28 Member States of the European Union.

LISIGE and INSPIRE demand public administrations to publish their cartography through standard web services.

3. Curating and visualizing the source data

In this section we will introduce the different sources of data that will be used for further statistical analysis. We will also explain the steps performed to obtain the curated data to be used as input for our predictive analysis.

3.1. Introduction to the different data sources

We will analyze three sources of data:

3.1.1. Fastly logs for user requests

The CartoDB platform uses the Fastly CDN service for all tile requests, as well as the requests of the interactive UTFgrids. These requests to all CDN nodes are stored in logs. These logs contain information about the requested resource, allowing to detect which map has been accessed and which of its tiles have been served. They also provide temporal information (when the request was performed) and the IP address of the source of the request, which allows tracking the behaviour of an individual over a single visualization.

3.1.2. Redis dump for maps metadata

The Redis key-value database stores the CartoCSS styles that define each map, as well as its interactivity options. This is the data that can be obtained from the original data source through browsing the visualization. This information allows us to match the requests obtained through the logs with the individual metadata of the maps that have been accessed.

3.1.3. PostgreSQL dump for map and layers metadata

The PostgreSQL dump contains all the information available for a map. To simplify the retrieval of data, some characteristics of the map will be obtained from the Redis cache contents. Other specifics such as the bounding box in which a map is located, its center or its level of zoom are available from Postgres. In the

PostgreSQL dump section below we will study and extract the contents of these tables for the posterior analysis.

3.2. Analysis of requests through CDN logs

In this section we will analyze the logs generated by Fastly's CDN as a source of data. We will cover the structure of the stored log requests and the meaning of each of its attributes. The syntax of each line generated in the logs that corresponds to a unique request is formatted as follows:

```
<ServerID> Timestamp (YYYY-MM-DDTHH24:MI:SSZ) NodeID Log-name [LogID]: Source-IP "-" "-" Timestamp (Day, DD Mon YYYY HH24:MI:SS GMT) GET (resource) (MISS/HIT) HTTP-status-code
```

Figure 3. a): Log event example

The most useful information that we can find in these events is:

- The timestamp of the request, showing when the request occurred
- The source IP address of the request, identifying different user activities
- The resource that was requested, such as a single tile or an UTFGrid object containing interactive information

By studying the time when the event took place in combination with the source IP address we can track the activity of individuals over a specific map. As an example, we can check the different resources requested by a single person in a specific temporal sequence to understand if the user was moving the map, zooming in/out or interacting with it by clicking or hovering its geometric features.

The resource parameter of the log event is the part that will give us most of the information. With a request URL we can determine the map that was accessed and the specific tiles that were requested.

In order to process 6405 compressed log files (with a size of approximately 9.3 GB) we will use the following regular expression:

```
    <\d+>
    (\d+-\d+-\d+T\d+:\d+:\d+Z)          # log timestamp
    \ (?:.*?)\[\d+\]:
    \ (\d+.\d+.\d+.\d+)                 # IP address
    \ (?:.*?)\ GET
    \ /([\w-]+)/+api/+v1/+map/+         # user + endpoint
    (?:([\w-]+(?:@|%40)\w+)(?:@|%40))?  # named map template (optional)
    (\w+):(\d+(?:\.\d+)?)/              # layergroup ID + timestamp
    /?
    (-?\d+)(?:,*\d*)*/                  # zoom (or layer number if torque)
    (-?\d+)/                            # x (or z if torque)
    (-?\d+)                             # y (or x if torque)
    (?:/-?\d+)?                         # (y if torque)
    \.([\w\.]+)                         # extension type
```

Figure 3. b): Regular expression for log processing

The groups in each match of the regular expression will be stored in an *Event* object that will be represented by the properties defined in the next figure:

```
Event = namedtuple("Event", [
    'x', 'y', 'z',
    'time', ‘user',
    ‘ip_address',
    'named_map_template',
    'layergroup',
    'layergroup_timestamp',
    'type'
])
```

Figure 3. c): Event object definition

The Python code that generates *Event* objects from the data can be found at *Appendix 12. A: Python script for the curation of log files*.

Log files are compressed in the BZ2 format. One of the advantages of this choice is that .bz2 files, unlike other compression methods, can be concatenated and still build a valid .bz2 file. This makes it easy to join together several files that, for example, might belong to the same data range but come from different CDN nodes.

Once that the data has been retrieved from the logs with the method explained above we will have a list of Event objects to analyze.

One of the most descriptive properties of an event is its type, which can be one of:

- **"png"**: the event corresponds to a request of a tile (PNG image)
- **"grid.json"**: the event corresponds to a request of interactive data — the user performed any cursor interaction in the map: hover or click
- **"json.torque"**: the event corresponds to a specific data format request that contains an extra temporal component

Torque[29] is an Open source library that allows rendering time-series data in the client side, instead of being rendered by a tiler server. Torque is outside of the scope of our work, hence the events corresponding to a request of type *json.torque* will be skipped for our tile analysis.

Other important properties that we are matching and storing are:

- **X, Y, Z**: parameters that define the exact tile that has been requested within the quadtree tile schema
- **time**: timestamp for the request
- **user**: username of the owner of the map
- **ip_address**: the source IP address from which the resource was requested
- **named_map_template**: if the map contains private data, this is its template identifier
- **layergroup**: alphanumerical string that uniquely identifies a map
- **layergroup_timestamp**: timestamp of the last edition operation on the map

We will group the different *Event* objects in a *Dataset* object.

By using a *Dataset* we will be able to start performing some easy queries and calculations. For example, we can obtain the subset of the requests that correspond to tile requests out of the whole *Dataset* by checking the "type" attribute of an event. In this scenario, the type of the request will be "png" if it corresponds to a tile request (an image).

3.3. Analysis of map metadata

In this subsection we will analyze the map metadata by using two different data sources: a Postgres database dump and a Redis dump.

3.3.1. Analysis of map metadata through a PostgreSQL dump

Our data source in this case is a 10.07 GB SQL file that we will handle in a personal installation of PostgreSQL.

We create a local database (named *tiledb*) and load all the data in the PostgreSQL dump by running the following commands:

```
psql create database tiledb
psql tiledb < map_metadata_dump.sql
```

Figure 3. d): PSQL commands for database generation

Once the database is rebuilt, we will have four tables to study: *maps*, *visualizations*, *layers* and *layers_maps*. The structure and information of all of them can be seen by using the command "`\d+ <tablename>`" in the psql[30] console, and it is as follows:

- **Maps table**

Column	Type	Description
view_bounds_sw	Text	Coordinate that defines the south-west bound of the map
view_bounds_ne	Text	Coordinate that defines the north-east bound of the map
center	Text	Coordinates in which the map is centered
zoom	Integer	Zoom level applied to the map
id	uuid	Identifier of the map
user_id	uuid	Identifier of the user that owns the map

Table 5: "Maps" table structure

- **Layers table**

Column	Type	Description
infowindow	Text	Fields that are enabled in the click infowindows of the map
tooltip	Text	Fields that are enabled in the hover infowindows of the map
options	Text	Options of the layer
order	Integer	Order that the layer has inside the map
id	uuid	Identifier of the layer

Table 6: “Layers” table structure

- **Layers Maps table**

Column	Type	Description
layer_id	uuid	Identifier for a specific layer
map_id	uuid	Identifier for the map to which the layer belongs

Table 7: “Layers Maps” table structure

- **Visualizations table**

Column	Type	Description
name	Text	Name of the map
description	Text	Description of the map
type	Text	Type of the map
tags	Text[]	Tags of the map
created_at	Timestamp	Creation time of the map
updated_at	Timestamp	Last update time of the map
map_id	uuid	Identifier of the map
user_id	uuid	Identifier of the user that owns the map
id	uuid	Identifier of the visualization

Table 8: “Visualizations” table structure

Notice that the different identifiers that are common in the four tables will allow us to join the data even though it is stored in different tables of the database. To retrieve the final data from all the tables in the database as a workable set, we will export a joint query as JSON. The psql command `\o <output_filename>` allows exporting to a file the results of a SQL query.

In order to obtain the output in the format we need, we will use the `array_to_json` Postgres function. By using this function in combination with `array_agg` (which appends each returned value inside an array) we will generate the final dataset. We will however need an alias for the intermediate results, which we will then aggregate and finally export as JSON. The final SQL statement is as follows:

```
WITH t AS (
  SELECT
  distinct(a.id) AS map_id,
  a.zoom,
  a.center,
  a.view_bounds_sw,
  a.view_bounds_ne,
  b.id AS viz_id,
  d.infowindow,
  d.tooltip,
  CASE
    WHEN d.options LIKE '%marker-width%'
      THEN 1
      ELSE 0
    END AS point,
  CASE
    WHEN d.options LIKE '%polygon-fill%'
      THEN 1
      ELSE 0
    END AS polygon,
  CASE
    WHEN d.options NOT LIKE '%marker-width%' AND d.options NOT LIKE '%polygon-
fill%'
      THEN 1
      ELSE 0
    END AS line
  FROM
  maps as a,
  visualizations as b,
  layers_maps as c,
  layers as d
  WHERE
    b.map_id = a.id AND a.id = c.map_id AND c.layer_id = d.id
  )
SELECT array_to_json(array_agg(t))
FROM t;
```

Figure 3. e): SQL statement for metadata obtention

The output file from this SQL statement will be named *maps_db_metadata.json*. It has a size of 423MB and contains JSON-like formatted text.

The inner structure of each object is the one defined by the selected columns in the SQL statement. They can be seen in the following figure:

```
[{
    ‘map_id',
    'zoom',
    ‘center',
    ‘view_bounds_sw’,
    'view_bounds_ne',
    ‘viz_id',
    ‘infowindow’,
    ‘tooltip',
    ‘point’,
    ‘polygon’,
    ‘line'
}]
```

Figure 3. f): Postgres data as a JSON

3.3.2. Analysis of map metadata through a Redis dump

Similarly as with the PostgreSQL dump, we will convert a Redis dump into JSON to make it easier to work between the different data sources. The Python utility *RDBtools*[31] is able to parse Redis *dump.rdb* files and export their data as JSON.

In order to load a Redis dump into our system, we can replace the default *dump.rdb* file that the Redis server generates the first time it is set up with the dump we want to load. Once this dump is loaded, we can select the keyspace in which the data is stored (the default keyspace is `db0`) and start loading the keys we need to obtain values from.

In our specific case we are interested in the *map_cfg* and *map_tpl* sets of keys, which contain the information of each layergroup and template ID. As an example, listing the value of all pairs for the *map_tpl* key is done as follows:

```
keys map_tpl|*
```

Figure 3. g): Redis command to list the set of map_tpl key-values

A challenge that we find with a datastore like Redis is that the contents on it get deleted after some time, defined by the "EXPIRE" flag of each independent key. If this flag is not overwritten or removed, loading an old Redis dump will probably generate an empty database if all the records have expired at that time.

By using a tool to export the Redis information into JSON we can work around this behaviour. After installing the *RDBtools* utility in our system, we can generate the output files by running the following commands:

```
rdb --command json redis_dump.rdb --db0 --key "map_tpl\|*" > tpls.json
rdb --command json redis_dump.rdb --db0 --key "map_cfg\|*" > cfgs.json
```

Figure 3. h): RDBtools command for exporting data to JSON

As a result of this operation, two files are generated: *tpls.json* (which contains information for 79571 different keys and has a size of 741MB) and *cfgs.json* (which contains information for 7180 keys and has a size of 17MB).

3.4. Visualizing the source data

In the following subsections we will start visualizing and getting insights from the data we have curated in the previous process.

3.4.1. Maps from requested tiles

In the same way that maps are used to represent geographical information, we will use them to have visual information about the tiles that are being requested.

With this objective, we have coded three drawing functions:

- `draw_basemap`: by providing an XYZ URL of a basemap service and a level of zoom, this function retrieves the different tiles and places them together to build a basemap image
- `draw_multimap`: this function gets as parameters a set of tiles and, optionally, a set of events, such as the most requested tiles, which are highlighted in the final image over the regular tile requests

- `draw_layergroup`: this function receives as parameters a complete set of tiles and a specific layergroup, analyzes the most requested events and plots the information for a specific layergroup

The source code of these functions and the helper function `most_requested_events` is available in *Appendix 12.B*.

The most accessed maps (or *layergroups*) represented as a map of tile requests can be found in the next figures.

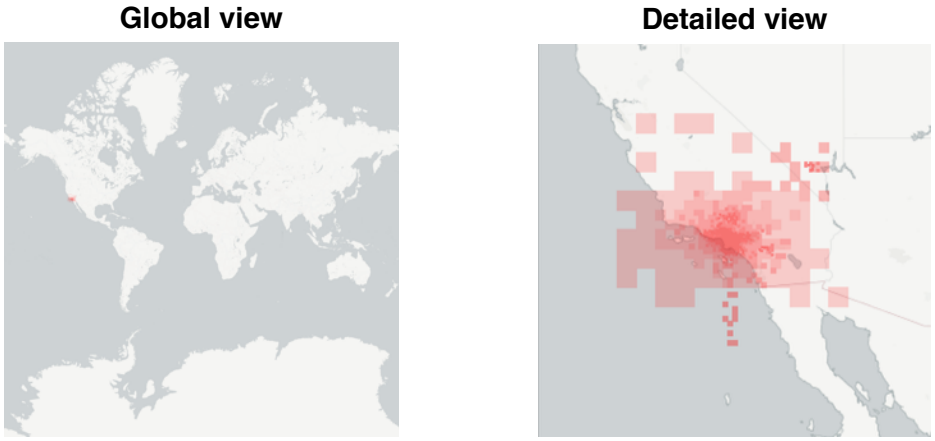


Figure 3. i): Global and detailed view of map d200f8c2d8a0fd4adffd6e8512576285

- **Day with most accesses:** June 6th, 2015 (19369 accesses)

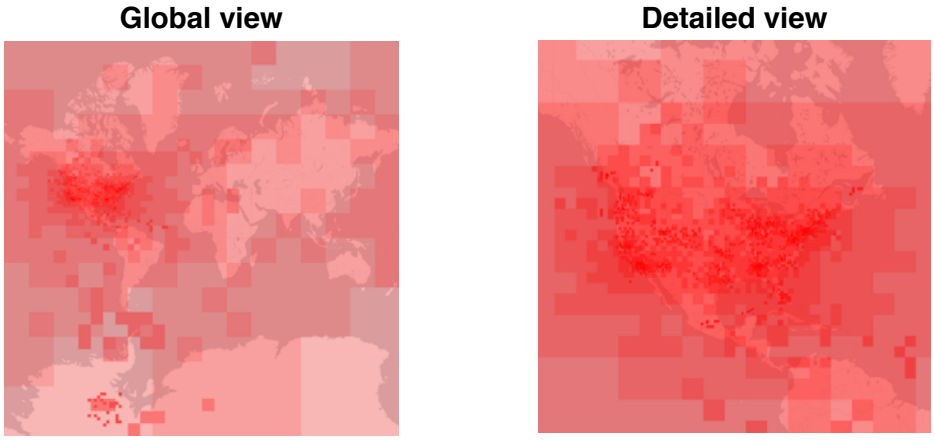


Figure 3. j): Global and detailed view of map f148fd4dad5313505f19b8032dfe2c6d

- **Day with most accesses:** June 19th, 2015 (16309 accesses)

Global view **Detailed view**

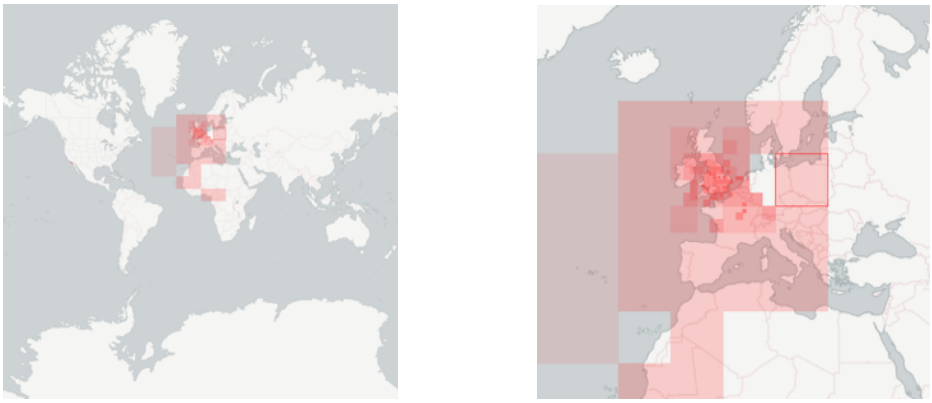


Figure 3. k): Global and detailed view of map dbcd5e1fdae10d922a66ebde44a35751

- **Day with most accesses:** June 27th, 2015 (12159 accesses)

Global view **Detailed view**

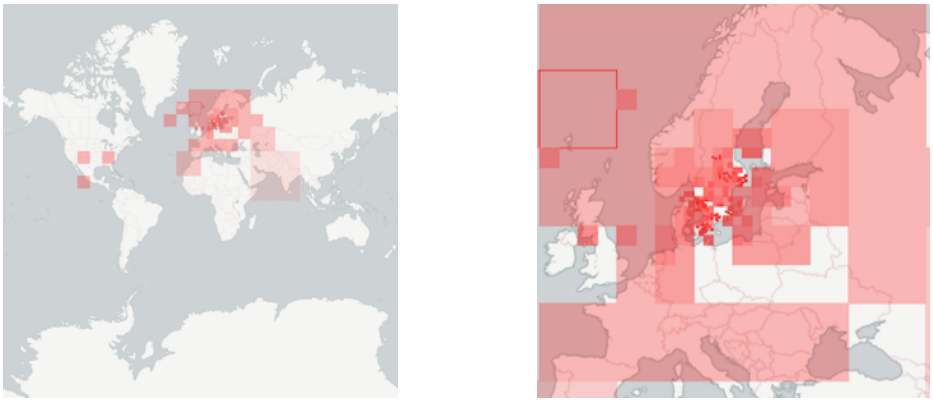


Figure 3. l): Global and detailed view of map 7f0d1c82b45abde157e8c5ff03a2b9fe

- **Day with most accesses:** July 2nd, 2015 (10321 accesses)

Global view **Detailed view**

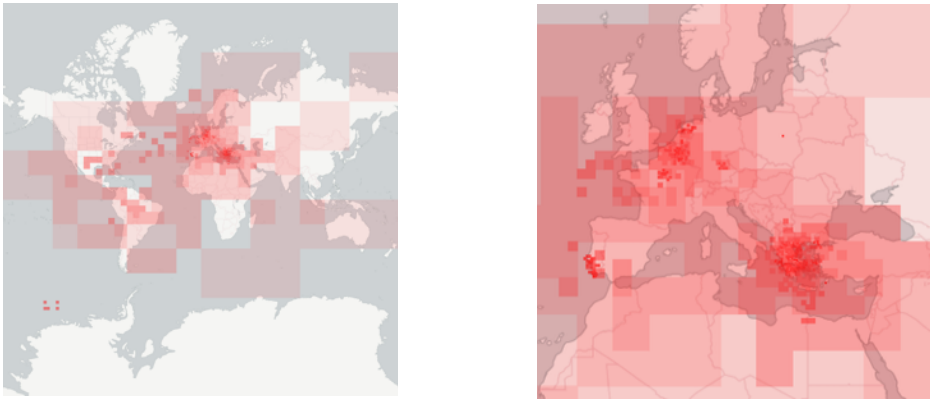


Figure 3. m): Global and detailed view of map a9938a1c41ffe4d4cb4ce6a5474e86ae

- **Day with most accesses:** June 16th, 2015 (10121 accesses)

3.4.2. Obtaining reference and test data

Log data is divided in several thousand files. Our goal is to reorganize all the data into two subsets: a reference subset we will use to design our heuristics and a test subset we will use to verify the behaviour of the heuristics. To do this, we have the following functions:

- `aggregate_data_files`: this function receives from its parameters a set of *.bz2* formatted files and a name, and concatenates them into a bigger .bz2 file with the specified name
- `get_sliced_data`: this function receives from its parameter a file set and the slicing mode (*reference* or *test*) and returns a file set containing a specific percentage over the total amount of files. By default, it uses 80% for reference data and 20% for test data

The source code of these functions and the helper function `get_filepaths` that allows getting a set of files that have a specific name is available in *Appendix 12.C*.

By following this approach we will generate two daily files, one for reference purposes and another one for testing. In total, we will work with 60 final files.

3.5. Statistical insights of the source data

These are the most important variables that we will be working with:

- Zoom level
- Bounding box
- Interactivity
- Geometry type

Besides these variables, we will also work with IP addresses (to distinguish individual behaviours) or time information. Also, thanks to the layer information that we obtain from the PostgreSQL dump, we will be able to compute the spatial distribution of the information shown in a map.

With the objective of handling this big amount of data in an efficient way, we will use Python iterators by importing the `itertools`[32] module, which is intended to be fast and use memory efficiently.

3.5.1. Events statistics

Firstly, we will analyze the global information as a series of counts and unique counts: For example, to obtain a total count of successful requests over a given time period. Similarly, getting a unique count of a given field will give us visibility into the number of unique accesses from user IPs.

The results of the analysis above can be found in the following table:

Date	Events	Tile requests	Unique accesses	Unique maps
June 06, 2015	46.808	38.488	4.126	1.185
June 07, 2015	98.775	80.672	12.346	2.259
June 08, 2015	49.119	38.371	3.417	1.449
June 09, 2015	30.695	32.709	3.209	1.518
June 10, 2015	76.197	62.376	11.059	1.666
June 11, 2015	60.953	48.842	5.299	1.908
June 12, 2015	52.606	41.995	5.580	1.429
June 13, 2015	46.110	35.884	4.458	1.261
June 14, 2015	54.683	38.321	5.524	1.244
June 15, 2015	56.449	45.023	5.080	1.592
June 16, 2015	64.595	52.081	5.448	2.188
June 17, 2015	64.621	51.950	5.858	3.055
June 18, 2015	58.685	45.009	7.054	1.896
June 19, 2015	65.687	51.912	5.374	1.786
June 20, 2015	53.314	43.338	4.953	1.234
June 21, 2015	39.039	30.432	3.961	1.104
June 22, 2015	46.306	35.899	3.400	1.492

Date	Events	Tile requests	Unique accesses	Unique maps
June 23, 2015	47.024	36.283	3.770	1.665
June 24, 2015	56.598	45.536	5.090	1.744
June 25, 2015	56.581	46.010	4.711	2.104
June 26, 2015	60.563	50.140	3.778	2.354
June 27, 2015	35.452	27.086	1.893	1.069
June 28, 2015	36.353	28.009	2.245	675
June 29, 2015	49.051	38.012	2.170	1.105
June 30, 2015	50.087	39.794	2.377	1.368
July 01, 2015	129.855	107.132	8.605	1.618
July 02, 2015	88.812	70.025	7.660	1.689
July 03, 2015	75.072	59.877	6.019	1.736
July 04, 2015	53.093	42.671	6.562	1.245

Table 9: Counting daily data from the logs

The total counts for the events and tile requests is then as follows:

Amount	Events	Tile requests
Total	1.703.183	1.363.877

Table 10: Total monthly counts

From this information we can obtain the average values of events per day, events per map, and events per user:

	Per day	Events per map	Events per user
Average	58.730,45	36,52	11,28

Table 11: Event averages

3.5.2. Interactivity statistics

With the information that we have for tile requests over the general events count, we can know how many requests were due to tile requests and how many were due to interactivity requests:

	% tile requests	% interactivity requests
Events	80,0781	19,9219

Table 12: Percentages of tiles and interactivity

On a daily basis, tile requests involve 76 to 83% of the total requests.

3.5.3. Zoom statistics

From a total of 28379 unique maps accesses during the period that we are studying, the statistics of the levels of zoom can be found in the following table:

Zoom level	Percentage	Zoom level	Percentage	Zoom level	Percentage
0	0,2180700	**10**	0,0599668	**20**	0,0029722
1	0,0750568	**11**	0,0531987	**21**	0,0008205
2	0,0316654	**12**	0,0537777	**22**	0,0000426
3	0,0357503	**13**	0,0451301	**23**	0,0000385
4	0,0415465	**14**	0,0454693	**24**	0,0000298
5	0,0461388	**15**	0,0375655	**25**	0,0000229
6	0,0454280	**16**	0,0259762	**26**	0,0000257
7	0,0554322	**17**	0,0211327	**27**	0,0000167
8	0,0389847	**18**	0,0127801	**28**	0,0000197
9	0,0479590	**19**	0,0047921	**29**	0,0001892

Table 13: Zoom level statistics

The real distribution of zoom levels among the 28379 analyzed maps obtained from the PostgreSQL dump data is as follows:

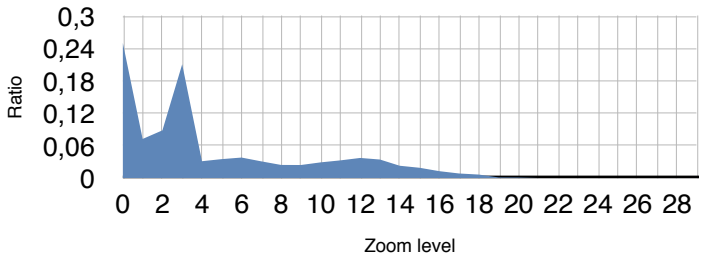


Figure 3.n): Zoom level distribution

3.5.4. Geospatial statistics

In terms of location, the mean of the set of coordinates in which all maps are centered is (-7.9157, 30.5698). Taking into account the previous statistics on the zoom levels, this average value makes sense: for low level of zooms (such as from 0 to 3) almost all the globe region is covered and traditionally, in the Mercator projection, the center is set at the Atlantic Ocean. This location can be seen in the following figure:

Figure 3.o): Average center coordinates map

The actual distribution of the center points of the complete set of maps can be seen in the following figure:

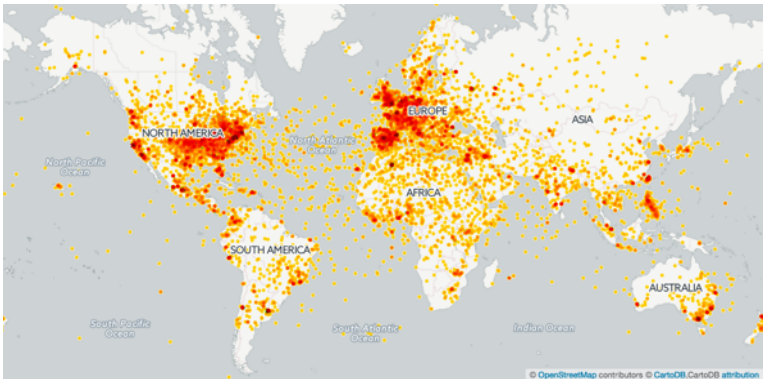


Figure 3.p): Distribution of the center point of the maps

In terms of areas, the regions mapped over the whole month that we are analyzing are represented in the following figure. This map has been generated by using the coordinates available in the PostgreSQL maps table and the `ST_MakeEnvelope` PostGIS function[33], which creates a rectangular polygon formed from the given minima and maxima.

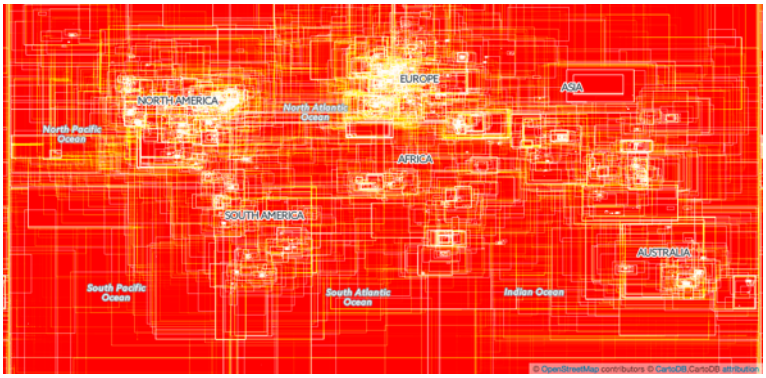


Figure 3.q): Representation of the bounding boxes of all maps

3.5.5. Data statistics

From the data point of view, the 28379 analyzed maps are formed by 51082 different layers. The different types of geometries are distributed within these layers as the following table indicates:

	Points	Lines	Polygons
Percentage	0,53099	0,21913	0,24987
Total amount	27.124	11.194	12.764

Table 14: Statistics of geometry types over all map layers

The Python script to retrieve simple statistics from log data can be found in *Appendix 12. D.*

4. Designing heuristics for tile caching

In the previous section we defined the methods for event processing by means of log crunching. Now, we are going to analyze these results in order to design the rules to follow for caching the tiles that are likely to be requested.

Slicing our analysis and grouping our data by different parameters will allow us to get a more granular view of the key information. Along this section, we will be treating each of these components in different subsections.

4.1. Zoom level and bounding boxes

The most important variable when it comes to understanding how users will interact with a map is the initial status of the map itself. The initial status of a map is defined by its level of zoom and its center point. Usually, these two variables are set so that the whole data is visible in this initial state.

Talking of a very small bounding box is synonym with saying that the map is focused in a high level of zoom. In order to cover a specific region of the Earth, there is a minimum level of zoom that applies for it. Because of this close relationship, we will be treating these two variables in the same section.

In *Section 3* we got some insights about the most used levels of zoom. For map initial states, the most used levels are from 0 to 3 (the level of zoom 0 is set in 25% of the maps). For requests, the most requested level of zoom is 0 (21%) while the rest of levels, until 15, are requested around 5% of the time each.

Different levels of zoom might trigger different behaviours in the interaction of a user with a map. If a user accesses a map which contains information about a neighborhood in Barcelona (a level of zoom of 15 would be appropriate here), we know that they could zoom in/out or move along a specific area, but other requests are not likely to happen. In *Section 4.4*. we will strengthen these ideas by obtaining detailed information about map activity.

4.2. Data, distribution and its interactivity

The type of the data that is being shown in a map, its aggregation and its ability to provide extra information are also the key to define user activity. Points are the most polyvalent and simple geometric type. They keep their meaning in any zoom level, whereas lines or polygons are more affected by these changes because they lose their meaning as soon as the resolution of the map decreases.

The spatial distribution of the data is important too. If a global map is set in a low level of zoom but it shows information about the provinces in each country, users will zoom and browse around them to check the specific details. However, if this global map only contains information about continents that is easily readable at first sight for a low level of zoom, it might not engage the same level of user activity.

Interactivity could be another source of user engagement in a map. If a visualization offers information when its geometries are interacted with, its data will be consulted. Similarly as in the data distribution scenario, having more data available will trigger more events from the users that access maps as a way to retrieve geospatial information.

In the next section we will study individual activity related to these parameters to understand how much they affect map browsing.

4.2.1. Empty tiles

Throughout this document we have been focused on data standards, formats, storage and rendering, but most of the tiles that made up a map do not contain data at all. As tiles follow a quadtree hierarchy, the 4 children of an empty tile will always be empty tiles. Therefore, the number of empty tiles for the next level of zoom can be computed as 4^n, being n the number of empty tiles of the current level of zoom.

From our data sources we also know the bounding boxes of a map. Tiles outside a map bounding box are empty tiles, and so are their children.

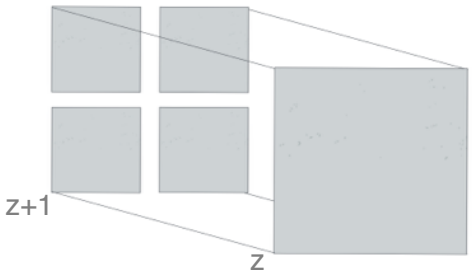


Figure 4.a): Empty tile quadtree representation

4.3. Creating caching rules through the study of the variables

Grouping log information by user and map will allow us to get a better understanding of the activity per user within each map. After the information is sliced, we will order it by timestamp in order to obtain a chronological feed of events that we will be able to map into common behaviours. The script to retrieve chronological activity grouped by users is available in *Appendix 12. E.*

For this, we will use just the 80% of the total log data we have as a reference. First, we will analyze how many requests for further levels of zoom are performed on average for each level of zoom. We note that statistically speaking, users always zoom in on the contents of the map; even if they did not, the process of zooming out would serve mostly tiles which are already cached. We disregard all zoom out requests from our analysis.

Zoom level	Avg. zoom requests	Zoom level	Avg. zoom requests
0	1,416	**5**	2,124
1	1,323	**6**	1,843
2	1,832	**7**	3,650
3	3,246	**8**	3,215
4	2,876	**9**	3,643

Zoom level	Avg. zoom requests	Zoom level	Avg. zoom requests
10	2,874	**20**	1,483
11	2,545	**21**	0,912
12	3,411	**22**	0,786
13	3,233	**23**	0,854
14	2,451	**24**	0,820
15	2,676	**25**	0,642
16	2,310	**26**	0,704
17	2,360	**27**	0,304
18	1,225	**28**	0,208
19	1,322	**29**	0,000

Table 15: Zoom level behaviours

To find the areas on which to focus our caching heuristics, we will calculate an information density ratio for each tile on the map. We define an information point as a point or a vertex (in the case of line or polygon geometries) from the dataset we are displaying on each map.

For tile *x,y* at zoom level *z* its density ratio will be equal to the number of information points on the tile divided by the average information points per tile for the current zoom level *z*. Because of this formula, tiles which are particularly dense will have a ratio above 1.0 whilst the least populated tiles will be below. This naturally causes our heuristics to emphasize the areas with the most information density.

We can calculate the density of points of a tile in PostGIS by clustering the data in regions of the size of a tile (256 pixels). In the following set of figures, we will show in terms of a scale of colors the amount of information points that lay in each

of the tiles. In this case, each one of the clusterized regions corresponds to the data from a Spanish municipalities dataset.

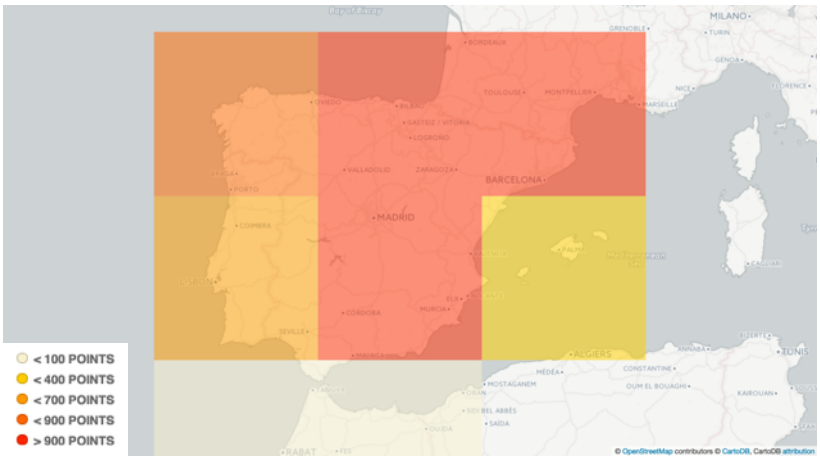


Figure 4.b): Point clustering density

As a reference, the source municipalities points map can be found in the next figure:

Figure 4.c): Spanish municipalities map

After performing some analysis, we are not detecting that the interactivity options of a map have an statistical relevance in a tile prediction scenario, so we will discard this variable in our final decision matrix.

4.4. Implementation details of the caching algorithm

Unfortunately, and due to constraints on the performance of our tile rendering process, it is not feasible to dynamically trigger caching jobs whilst the user

interacts with the map: our logs on user interactivity show that users spend on average less than 3s on each zoom level of the map. CDN log analysis, however, shows an average of 220ms rendering time spent per tile, allowing us to cache 13 tiles during the average time a user spends in any given zoom level. Given that the default display size of our maps on desktop platforms shows 15 tiles on the screen at a time, our dynamic caching would always lag behind the actual interaction behavior of the user.

Because of this, our practical implementation will stick to optimistically caching the most likely visited tiles on map generation by running in an offline job on the server side, once and only once after map creation.

We need however to set a hard limit on the amount of time the background job can process and cache tiles, as to not affect the overall performance and availability of our platform. Following existing patterns in our infrastructure, we will limit the total runtime of the background job to 60s in all cases, resulting in the capacity to cache 272 tiles on average.

The question is now how to choose the up to 272 tiles that will be cached from the potentially huge set of data on the map. Here is where our data analysis will come into play, aiding us to design a viable and accurate heuristic.

For our caching algorithm, we will exploit the properties of quadtrees: tile selection will operate as a depth-first traversal of the quadtree, beginning at the initial zoom level of the map, and culling its trajectory based on the data density of each individual tile and the expected zoom level that a user would reach. This simple heuristic, as we will see later on, is trivial to implement and provides great results when applied to real world data.

In practice, the traversal algorithm will not run until a fixed amount of tiles are cached (e.g. 272), but until the timeout for the background job is hit, which means than in some highly dense maps we will not be able to cache as many tiles as on simpler ones.

4.5. The walk through the quadtree

The depth-first traversal of the quadtree cannot be performed naively, as that would mean focusing all our caching efforts into a narrow view of the map (i.e. one that eventually reaches the maximum zoom level for the first quad we descended into). Because of this, we will cull the depth of the walk at a limit based on the amount of zoom that an user is expected to perform from a given zoom level (as previously studied on Table 16).

This culling however will be biased based on the expected amount of information density in that branch of the quadtree. Likewise, the order of the branches of the quadtree will also be influenced by the expected amount of information density.

Considering this, we can define our heuristic with this Python pseudo-code:

```
def caching_traversal(quads, max_depth):
    if max_depth <= 0 or timed_out():
        return


    quads = sorted(quads, key=lambda q: q.information_density)
    for q in quads:
        cache_tile(q)
        md = (max_depth - 1) * q.information_density
        caching_traversal(q.quads, md)


caching_traversal(quads, AverageZoom[z])
```

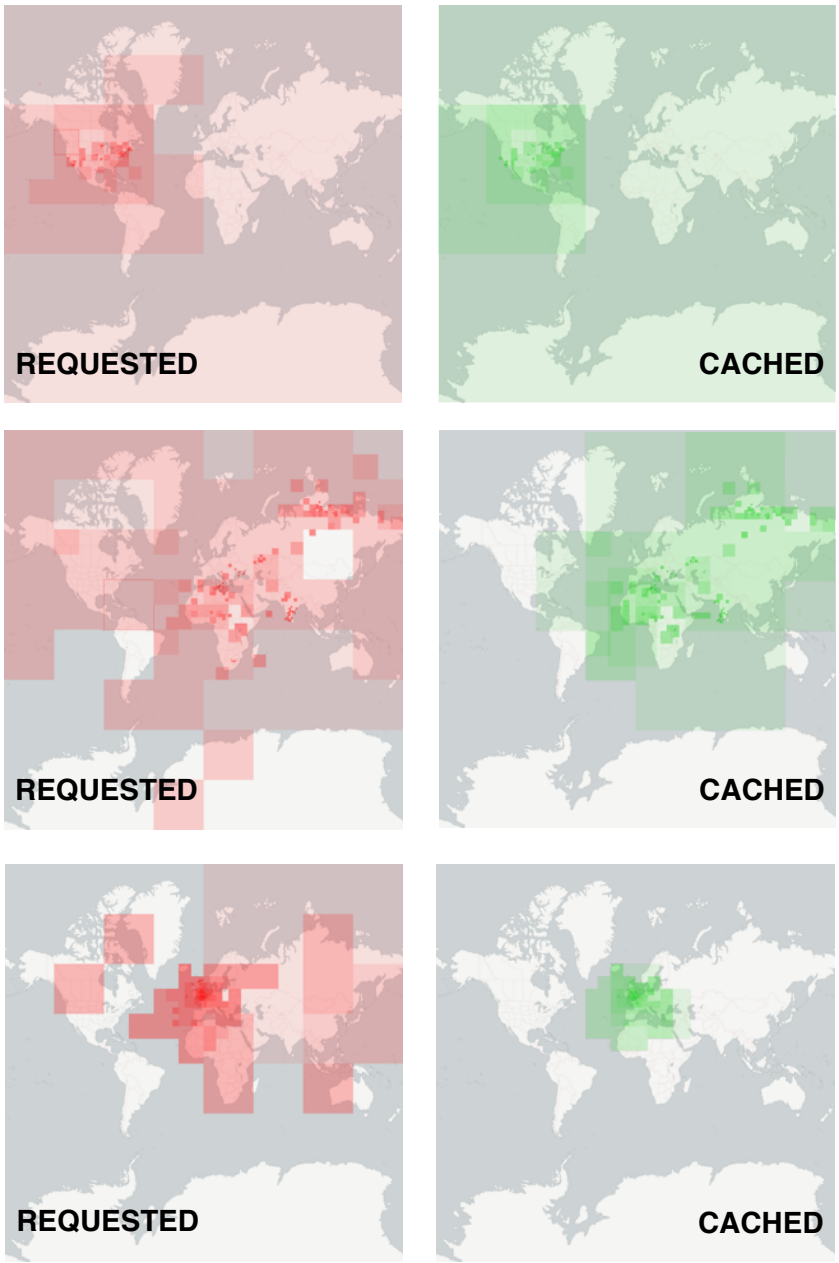
Figure 4.d): Traversal caching pseudo-code

The caching traversal begins at the initial set of quads for the map and will descend as many zoom levels as the average values seen in Table 16.

For each level of recursion, we will decrease the depth accordingly but we will also adjust it based on the information density ratio for the quad we are descending into. This way we make sure that the most dense quads will be cached more frequently than their simpler counterparts.

4.6. Testing caching heuristics results

To verify our heuristics, we will study various interaction patterns in our test dataset by rendering them as we previously saw on *Section 6.1*. We will also tweak our caching algorithm as to *mark* the cached tiles on a blank basemap instead of actually caching them. With this, we will be able to compare the interaction patterns with the renderings from our caching heuristic and verify visually that the cached tiles are an useful and accurate subset of the actual interactions performed by our users.



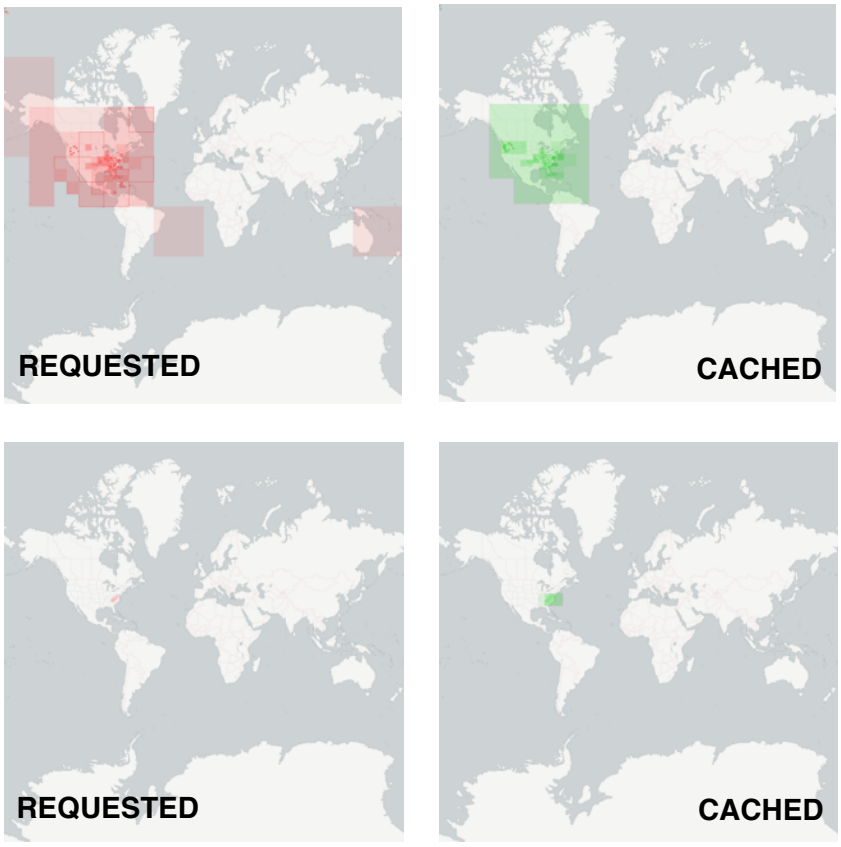


Figure 4.e): Request and cached tiles comparison

From the resulting images above, we see how our algorithm focuses the caching efforts in the tiles that contain information. The tiles that do not contain information are discarded, so the cached schemes show a more focused result than the requested schemes, which show outlying tiles.

Another detail that we can notice through the color of the cached schemes is that the zoom levels are not completely cached (this scenario would show a big aggregation of opaque regions). With these visual results we can conclude that our algorithm is accurate enough to significantly reduce the amount of tiles that need to be dynamically generated for the average user. In the following section we will analyze the costs of the tile generation process and the impact of our caching solution.

5. Real applications of cache generation

In this section we will study the benefits and effects of cache generation.

5.1. Render timing breakdown

The first step of the tile rendering process is to obtain the data that needs to be shown in the tile. This is done with a SQL query performed on the database where the geometric information is stored. We will refer to the time of this operation as T_{DATA}.

Windshaft, the Node.js tile renderer, uses several background threads to process rendering requests. A new rendering request will wait until any of these threads is not busy to start rendering the data. The amount of time that the process is queued waiting for a thread will be defined as T_{QUEUE}.

Finally, when the actual rendering process has started, the tiler spends a variable amount of time generating the tiles, which depends on geometry complexity and on the styles applied to the features. This amount of time will be defined as $T_{RENDERING}$.

Therefore, the time needed for rendering a tile (T_{TILE}) can be defined as follows:

$$T_{TILE} = T_{DATA} + T_{QUEUE} + T_{RENDERING}$$

We can consider that rendering processes are never blocked by busy threads. In this case, we can approximate T_{TILE} to:

$$T_{TILE} \approx T_{DATA} + T_{RENDERING}$$

5.2. Reducing the map generation time

Caching tile resources greatly improves response times of any interactive map when accessed by the user. In general, tile generation is an expensive process and caching allows reducing this cost to its minimum.

In this section we will study how the different components affect the total rendering time of a map. We will also measure by means of the *time to first byte* how the behaviour of a map improves when the contents are served from a cache.

5.2.1. Time analysis on geospatial complexity

The number of geometries that are to be drawn in a single tile or their precision (the number of vertices that define a line or a polygon) affect the rendering time. In the following examples we will analyze different use cases and their time:

- **Points. Low aggregation.**
  The time to render 9 points in a tile is:
  - T_{DATA}: 4 ms
  - $T_{RENDERING}$: 159 ms
  - T_{TILE}: 163 ms
  - T_{CACHED}: 48 ms
  - T_{TILE} - T_{CACHED}: 115 ms

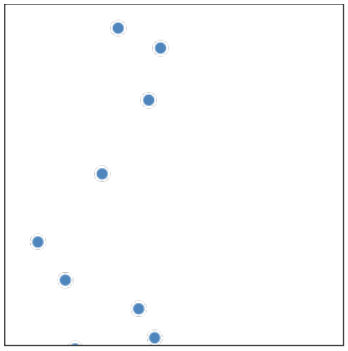


Figure 5. a): Low aggregation tile

- **Points. High aggregation.**
  The time to render 13553 points in a tile is:
  - T_{DATA}: 403 ms
  - $T_{RENDERING}$: 4.427 s
  - T_{TILE}: 4.83 s
  - T_{CACHED}: 57 ms
  - T_{TILE} - T_{CACHED}: 4.773 s

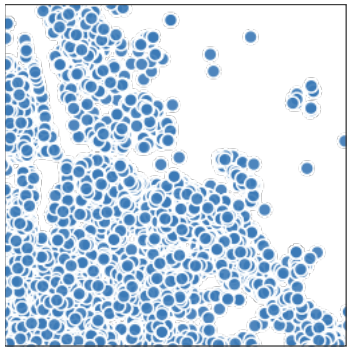


Figure 5. b): High aggregation tile

- **Polygons. Low precision.**
  The time to render 7119 polygon vertices in a tile is:
  - T_{DATA}: 8 ms
  - $T_{RENDERING}$: 167 ms
  - T_{TILE}: 175 ms
  - T_{CACHED}: 51 ms
  - T_{TILE} - T_{CACHED}: 124 ms

Figure 5. c): Low precision tile

- **Polygons. High precision.**
  The time to render 2988837 polygon vertices is:
  - T_{DATA}: 268 ms
  - $T_{RENDERING}$: 2.182 s
  - T_{TILE}: 2.45 s
  - T_{CACHED}: 50 ms
  - T_{TILE} - T_{CACHED}: 2.4 s

Figure 5. d): High precision tile

5.2.2. Time analysis on design complexity

The CartoCSS code that is applied to design a map can also add a variation in the rendering time of the tiles.

- **Polygons. Simple CartoCSS style.**
  The time to render a tile with a simple CartoCSS style is:
  - T_{DATA}: 43 ms
  - $T_{RENDERING}$: 137 ms
  - T_{TILE}: 180 ms
  - T_{CACHED}: 52 ms
  - T_{TILE} - T_{CACHED}: 128 ms

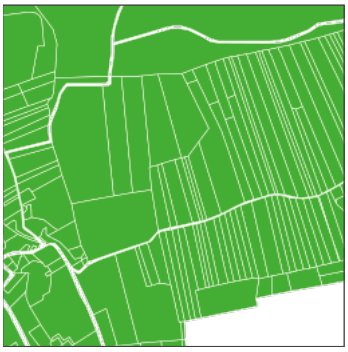


Figure 5. e): Simple design tile

The CartoCSS code applied to the features contain the most basic attributes for polygon geometries. It is presented in the following figure:

```
#layer{
  polygon-fill: #229A00;
  polygon-opacity: 0.8;
  line-color: #FFF;
  line-width: 0.5;
  line-opacity: 1;
}
```

Figure 5. f): CartoCSS simple code for design analysis

- **Polygons. Advanced CartoCSS style.**
  The time to render a tile by using an advanced CartoCSS style is:
  - T_{DATA} : 43 ms
  - $T_{RENDERING}$: 431 ms
  - T_{TILE}: 474 ms
  - T_{CACHED}: 49 ms
  - T_{TILE} - T_{CACHED}: 425 ms

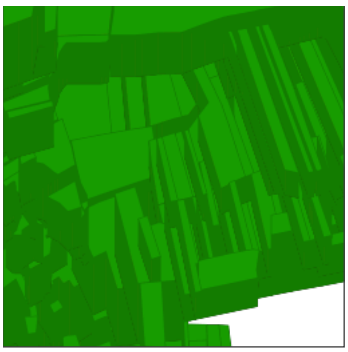


Figure 5. g): Advanced design tile

The CartoCSS code applied to the features contain "building" properties that allow drawing the polygons in a 3D shape. It is presented in the following figure:

```
#layer{
  building-height: [height]/10;
  building-fill:  #229A00;
  building-fill-opacity:1;
}
```

Figure 5. h): CartoCSS advanced code for design analysis

5.2.3. Time analysis on data computation complexity

When a map is created from data generated at runtime in a SQL statement, the response time of this request has an impact in the tile rendering total time:

- **Simple data generation.**
  The time to render a tile with data obtained from a simple SQL statement is:
  - T_{DATA} : 56 ms
  - $T_{RENDERING}$: 145 ms
  - T_{TILE}: 201 ms
  - T_{CACHED}: 52 ms
  - T_{TILE} - T_{CACHED}: 149 ms

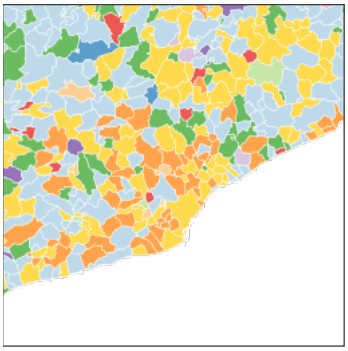


Figure 5. i): Simple SQL tile

In this scenario, we are using a table in which all the information is available. The CartoCSS code applied is a simple categoric one whose colors depend on a

value of the dataset.The SQL statement from which the data is obtained is presented in the following figure:

```
SELECT * FROM aggregated_data_table
```

Figure 5. j): SQL statement for simple data generation

- **Advanced data generation.**
  The time to render a tile with data obtained from an advanced SQL is:
  - T_{DATA}: 11.32 s
  - $T_{RENDERING}$: 301 ms
  - T_{TILE}: 11.621 s
  - T_{CACHED}: 59 ms
  - T_{TILE} - T_{CACHED}: 11.562 s

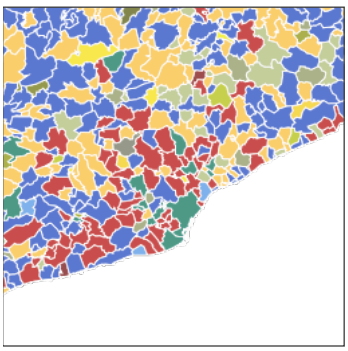


Figure 5. k): Advanced SQL tile

In this scenario, we are retrieving data from two different tables, processing the string in the results and using 3 filters. Similarly as the previous example, the CartoCSS code used is drawing the different polygons by using different colors that depends on their data.The SQL statement from which the data is obtained is presented in the following figure:

```
SELECT p.*, d.escrutado, replace(trim(upper(d.partido_ganador)),'#','') as
partido_ganador, d.p_votos, d.numero_votos, d.cod_municipio,
d.nom_municipio, d.numero_regidores, d.participacion

FROM poligonos_municipio as p, datos_elecciones_municipales as d
WHERE p.cod_municipio = d.municipio
AND p.cod_provincia = d.provincia
AND d.escrutado >= 3
```

Figure 5. l): SQL statement for advanced data generation

5.3. Protecting the mapping platform

Cache generation also has a positive impact on the stability of the whole system. By caching the tiles and avoiding them to be generated continuously we are saving a lot of resources in a platform.

6. Project planning and budget

Along this section we will discuss the structure and planning of the project. Then, we will study the costs that would be needed to put out solution into practice.

6.1. Project planning

The stages of the project and the temporal analysis of its activities will be studied in the following subsections:

6.1.1. Stages of the project

The project is divided in four main stages:

1. **Obtention and curation of the data:**
   The data analysed in this project has been obtained from logs offered for academic purposes by CartoDB, Inc, which collect the requests performed to public maps of their users. This raw data will be processed with regular expressions and transformed to objects usable in a script.

2. **Analysis of the data:**
   Once the source information is obtained and curated, its different characteristics will be studied in order to categorise the different kinds of data that are being used.

   This stage will present some statistics on the type of tiles that are requested and on the data from which the maps are created, such as the geospatial density or the bounding box of the datasets.

3. **Study of different statistical approaches to build a tile predicting model over the training data.**

4. **Application of the different statistical approaches and conclusions.**

6.1.2. Table of activities

The stages documented in the previous section 6.1.1. can be separated into the different activities presented in the table below. The duration of the activities is specified in hours of work and the total amount of estimated time for the 12 defined activities is 297 hours.

Description	Identifier	Requirements	Duration
Initial research on web mapping technologies	A	-	10
Obtention of data	B	-	1
Develop a method to curate the data	C	B	20
Obtain initial statistics and conclusions from the data	D	C	30
Analyse convenient statistical approaches to be applied	E	D	40
Apply and compare the different approaches	F	E	45
Implement predictive model	G	F	35
Test predictive model with test data	H	G	20
Obtain conclusions from the statistical results	I	H	20
Study economical impact of the solution	J	I	15
Report the process	K	A	60
Present the conclusions	L	K, J	1

Table 16: Table of project activities

Assuming a working day of 4 hours, the project would take 75 days, or two and a half months, to conclude. As most of the activities, except the one in charge of writing this report and the economical study of the solution, need to be performed sequentially the project could be considered as inflexible in terms of the critical path method.

6.1.3. PERT diagram

The activities defined in Table 16 are shown graphically in the following PERT diagram:

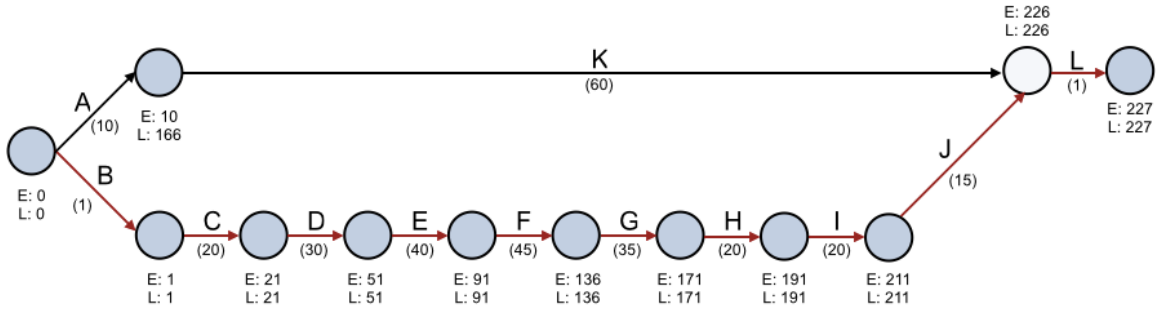


Figure 6. a): PERT diagram

The critical path of the project — marked in red in the diagram above —is formed by the sequence of activities B-C-D-E-F-G-H-I-J-L, implying that any delay in these activities will affect to the general project planning. The activity K ("Report the process") has a float of 156 days, meaning that it could be postponed until the day 166 in the schedule without implying any extra delay in the project. In reality, the activity K will be developed during the different stages that form the critical path as each one of them will need to be documented in the final report.

6.1.4. Gantt diagram

The temporal organisation of the activities in the project is presented in the following Gantt diagram. As in the section above, a day is considered as 4 hours and the total length of the activities is, approximately, 8 weeks including weekends.

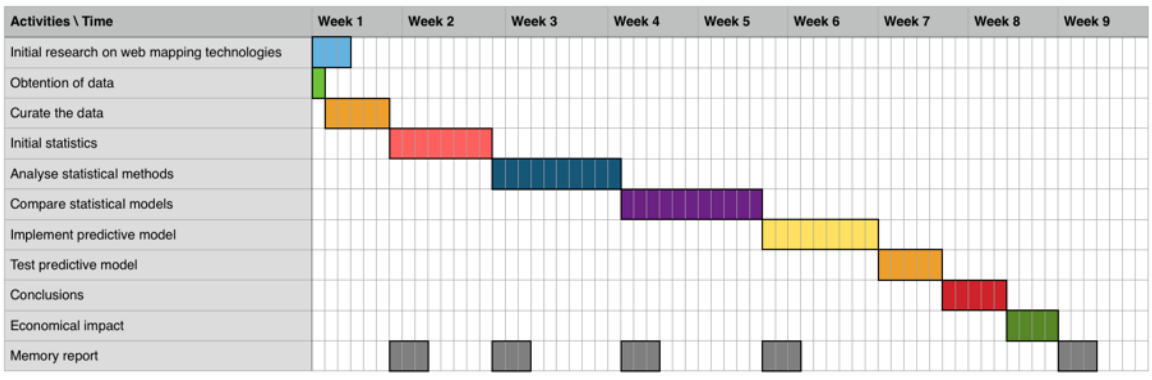


Figure 6. b): Gantt diagram

6.1.5. Tools

In this section we will cover the specifics of the tools and languages utilised in the project, grouped by utilities, as well as the reasons why those where chosen throughout the development of the cache generation solution. Along the scope of the document we will be studying how tiles are requested with different tools and methods described below.

- **Programming languages and libraries**

In order to manage the data the Python programming language will be used due to its ease when processing input data such as text logs, which will be our main source of information. The main reason why Python language was chosen was the existence of the different data analysis tool available for it, as the interactive computational environment IPython Notebook which are an advantage considering the nature of the project. The Pillow library will be presented and used in order to generate geospatial images from the collected data and to show the results as a visual output.

- **GIS applications**

In the area of web mapping and GIS applications, the CartoDB open source project will be used in order to generate map graphics.

Its architecture will be studied in *Section 2.2.: Technical overview of web mapping* and it will serve as an example of web mapping platform architecture. From now on and unless further specification, all mentions to “CartoDB" in the context of this document will be referred to the CartoDB open source platform — not to the commercial service from CartoDB, Inc.

- **Databases and caches**

In order to deal with the provided map metadata and process it in a usable format, Redis and Postgres databases will be used to retrieve an analyse the necessary information.

Redis is a key-value store — that can be used as a permanent database too — in which the data types are independent for every record, meaning that the values for different stored keys can have different types, fields or formats. Postgres is a relational database that provides special capabilities for storing geospatial data.

The specific details about how these different technologies are used in this project will be explained in *Section 3: Curating and visualizing the source data*.

6.2. Economic analysis on cache generation

Caching our resources in a content delivery network network implies paying the cost of this service. We will analyze the pricing of three content delivery companies: MaxCDN, Akamai and Fastly.

These companies usually charge per bandwidth and per number of HTTP and HTTPS requests received. The pricing table obtained from the pricing information of each service is as follows:

Pricing	MaxCDN	Akamai	Fastly
Bandwidth: 10TB	$499	$3.500	$1.200
Bandwidth: 100TB	$2.560	$35.000	$7.400
10.000 HTTP/HTTPS requests	N/A	N/A	$0,0086 (avg)

Table 17: CDNs pricing table

Assuming an average tile size of 40KB, we would serve 250 millions of tiles with a bandwidth of 10TB. Measuring it per each gigabyte of traffic, it would correspond to 25000 tiles.

We also have to take into account that some CDNs also charge for HTTP or HTTPS requests, as in the case of Fastly. The cost per 1000 tiles in the services mentioned above would be as follows:

Pricing	MaxCDN	Akamai	Fastly
For less than 100TB of bandwidth	$0,001996	$0,014	$0,0134
For a total BW greater than 100TB	$0,001024	$0,014	$0,01156

Table 18: Cost per 1000 tiles served by a CDN

While MaxCDN seems to be the most competitive option in terms of cost per served tile, most of its infrastructure is located in USA and Canada as shown in their network map[34].

Fastly also follows this pattern with 18 nodes in the USA and Canada region and only 4 in Europe, according to their network map[35].

The third option, Akamai, offers the same coverage on CDN nodes in the regions of USA, Canada, Europe and Asia, which could be a good service to use if we care about the transmission delays of our users in those regions.

7. Conclusions

Throughout this document we have studied the most important concepts of web mapping (covering from geospatial data formats to the different mapping libraries that put the tiles in place) as an introduction to the most technical details involved in tile generation and caching.

We have curated hundreds of gigabytes of data, with the goal of guiding the design and tests for our heuristics on the tiles that are likely to be requested depending on their geospatial properties.

Then, we have generated an algorithm to decide which tiles need to be cached supported the properties of quadtrees and the amount of information that exists beyond the different levels of zoom. After using our algorithm to test its efficacy, the outcomes are optimistic. Despite the fact that extremely dense maps will not be completely cached because of the big amount of tiles included in them at high resolutions, the regions that contain the most information will be covered. This happens because of the way we weight areas with higher information density in our heuristic.

By performing an analysis on tile rendering times, we have discovered that caching complex maps has a big impact on the time that a user will have to wait for them to be shown completely in a screen. This solution improves the response time of the maps while also saving rendering resources.

As a final step for our project, we have analysed the socioeconomic environment of our solution inside the web mapping market. In order to implement the solution in practice, we have listed the most common CDN services that will allow us to cache and serve the generated tiles and the expected costs.

8. References

[1] Wikipedia contributors. *Quadtree*. Wikipedia, The Free Encyclopedia https://en.wikipedia.org/wiki/Quadtree

[2] CartoDB, Inc. *The versatility of retrieving and rendering geospatial data with CartoDB*. May 16, 2013. http://blog.cartodb.com/the-versatility-of-retreiving-and-rendering-geospatial/

[3] Rogers, Simon. *Sunrise on Twitter - The animated map*. The Guardian. May 17, 2014. http://www.theguardian.com/news/datablog/ng-interactive/2014/may/07/sunrise-twitter-animated-map

[4] Elstein, Aaron. *Demand and opportunities soar for digital mapmakers*. Crain's New York. August 25, 2015. http://www.crainsnewyork.com/article/20150825/TECHNOLOGY/150829943/city-mapmakers-are-making-lots-of-money

[5] *Open Geospatial Consortium* (OGC): http://www.opengeospatial.org/

[6] Open Geospatial Consortium Inc. *OpenGIS® Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture*. May 28, 2005. http://portal.opengeospatial.org/files/?artifact_id=25355

[7] Aven, Nicklas; Moullet, Cedric; Ramsey, Paul. *Tiny Well-Known Binary Specification*. https://github.com/TWKB/Specification/blob/master/twkb.md

[8] Wikipedia contributors. *Shapefile format*. Wikipedia, The Free Encyclopedia https://en.wikipedia.org/wiki/Shapefile

[9] Wikipedia contributors. *Keyhole Markup Language format*. Wikipedia, The Free Encyclopedia https://en.wikipedia.org/wiki/Keyhole_Markup_Language

[10] Butler, Howard; Daly, Martin; Doyle, Allan; Gillies, Sean; Schaub, Tim; Schmidt, Christopher. *The GeoJSON format specification*. June 16, 2008. http://geojson.org/geojson-spec.html

[11] JavaScript Object Notation. http://www.ietf.org/rfc/rfc4627.txt

[12] Olaya, Víctor. *El libro GIS*. March 24, 2011. ftp://ftp.ehu.es/cidira/profs/iipbaiza/Libro_SIG.pdf

[13] Open Geospatial Consortium Inc. *OpenGIS® Implementation Standard for Geographic information - Simple feature access - Part 2: SQL option*. August 04, 2010. http://portal.opengeospatial.org/files/?artifact_id=25354

[14] *PostGIS*. http://postgis.net/

[15] Boundlessgeo. *Introduction to PostGIS*. August 04, 2010. http://workshops.boundlessgeo.com/postgis-intro/introduction.html

[16] *PostgreSQL: the geometric datatype* http://www.postgresql.org/docs/8.2/static/datatype-geometric.html

[17] *PostgreSQL: functions for geometric datatypes* http://www.postgresql.org/docs/8.2/static/functions-geometry.html

[18] *Mapnik*. http://mapnik.org/

[19] *CartoCSS*. https://github.com/mapbox/carto

[20] *Windshaft*. https://github.com/CartoDB/Windshaft

[21] *UTFGrid specification*. https://github.com/mapbox/utfgrid-spec

[22] Käfer, Konstantin. *How interactivity works with UTFGrid*. https://www.mapbox.com/blog/how-interactivity-works-utfgrid/

[23] *Leaflet*. http://leafletjs.com/

[24] *CartoDB.js*. https://github.com/CartoDB/CartoDB.js

[25] *Mapbox.js*. https://www.mapbox.com/mapbox.js

[26] Kurose, Jim; Addison-Wesley, Keith Ross. *Computer networking: a top down approach featuring the Internet, 2nd Edition*.

[27] Spain. *Ley 14/2010 sobre las infraestructuras y los servicios de información geográfica en España* . http://www.idee.es/web/guest/espanol-lisige

[28] European Union. *Directive 2007/2/EC of the European Parliament and of the Council of 14 March 2007 establishing an Infrastructure for Spatial Information in the European Community*. http://inspire.ec.europa.eu/

[29] *Torque*. https://github.com/CartoDB/torque

[30] *PSQL Interactive Postgres Terminal*. http://www.postgresql.org/docs/9.0/static/app-psql.html

[31] *RDBtools*. https://github.com/sripathikrishnan/redis-rdb-tools

[32] *Itertools*. https://pymotw.com/2/itertools/

[33] PostGIS contributors. *PostGIS ST_MakeEnvelope function*. http://postgis.net/docs/ST_MakeEnvelope.html

[34] *MaxCDN Network Map*. https://www.maxcdn.com/features/network/

[35] *Fastly Network Map*. https://www.fastly.com/network

9. Terminology

***AJAX* (Asynchronous JavaScript and XML).** Techniques used on the client-side of web applications that allow sending and retrieving data from a server without interfering with the display and behavior of the existing page

ArcGIS. Geographic information system application for working with maps and geographic information, developed by ESRI.

Basemap. Non-editable layer that provides background to a map. It is typically designed to provide a visual reference for other data layers to help orient the user.

CartoCSS. Language for map design developed by Mapbox, similar in syntax to CSS but oriented to styling geometries.

CartoDB. GIS Open Source application that provides location intelligence and data visualization capabilities.

CDN (Content Delivery Network). Network of distributed servers that deliver web content to a user based on their geographic location.

ESRI. Company founded in 1969 that supplies GIS software and geodatabase management applications.

Fastly. Paid content delivery network service that offers a set of APIs and allows customisation.

GeoJSON. Format for encoding geographic data structures based on the JavaScript Object Notation (JSON) format.

Geometry. Representation of the spatial component of geographic features, such as lines, points or polygons.

GIS. Any system designed to capture, store, manipulate, analyze or present all types of spatial or geographical data.

Infowindow. HTML popup that is used in web mapping to show the attributes of a feature.

KML. File format used to display geographic data in a mapping application based on the XML format.

Leaflet. Open source JavaScript Library for mobile friendly interactive web maps.

Mapbox. Company founded in 2010 that develops software to create custom online basemaps.

Mapnik. Open source mapping toolkit for tile rendering, written in C++.

***OGC consortium*:** International voluntary consensus that encourages the development and implementation of geospatial open standards.

Open data. Data that can be freely used, re-used and redistributed by anyone without restrictions from copyright, patents or other mechanisms of control.

Open source (software): Software for which the original source code is made freely available and may be redistributed and modified.

OpenStreetMap. Free and editable map of the whole world that is being built by volunteers largely from scratch and released with an open-content license.

PostGIS. Open source extension for PostgreSQL that adds support for spatial functions and geometry data types to the database.

PostgreSQL. Open source object-relational database system.

Projection. Method used for representing a three-dimensional object like the Earth on a two-dimensional surface. A projection is defined by its SRID (spatial reference system identifier).

Psql. Interactive terminal for working with Postgres that allows to query, edit and manage a database.

QGIS. Free and open-source desktop geographic information system application that provides data viewing, editing, and analysis capabilities.

RDBtools. Python utility that is able to parse Redis dump.rdb files.

Redis. Open source and BSD licensed advanced key-value cache and store.

Shapefile. Vector data storage format for storing the location, geometry, and attributes of geographic features.

Tile. Image of 256x256 pixels that represents a fixed geographical region and makes up a map once combined with adjacent tiles of the same level of zoom.

Tilemill. Tool for cartographers to design maps for the web using custom data.

Torque. Open source library that allows rendering time-series data in the client side, instead of being rendered by a tiler server

UTFGrid. Invisible tile layer made up of indexes and data that is shown in the interactive options of a map, such as the information of a specific feature when a user clicks or hovers it.

UTM (Universal Transverse Mercator). Projection system that divides the Earth into sixty zones, each a six-degree band of longitude, and uses a secant transverse Mercator projection in each zone. It preserves angles and approximates shape but distorts distance and area.

Varnish. Web application that caches any contents in a server through the HTTP level.

Web Mercator (projection). Variation of the Mercator projection which is the de facto standard for Web mapping applications.

Well-known binary (WKB). Standard format used to describe geometries as a stream of bytes.

Well-known text (WKT). Standard format used to describe geometries in a human readable way.

Windshaft. Map tile server written in Node.js for PostGIS with CartoCSS styling.

10. List of figures

The list of the different figures presented in the different sections are as follows:

(1) Motivation and goals

(2) Introduction to web mapping

(3) Curating and visualizing the source data

11. List of tables

The list of the different tables presented along the different sections are as follows:

12. Appendix

12.A. *Python script for the curation of log files*

```
# -*- coding: utf-8 -*-
import re
import datetime
import itertools
import random
import os
from collections import namedtuple


LINE_REGEXP = re.compile("""
    <\d+>
    (\d+-\d+-\d+T\d+:\d+:\d+Z)          # log timestamp
    \ (?:.*?)\[\d+\]:
    \ (\d+.\d+.\d+.\d+)                 # IP address
    \ (?:.*?)\ GET
    \ /([\w-]+)/+api/+v1/+map/+         # user + endpoint
    (?:([\w-]+(?:@|%40)\w+)(?:@|%40))?  # named map template (optional)
    (\w+):(\d+(?:\.\d+)?)/              # layergroup ID + timestamp
    /?
    (-?\d+)(?:,*\d*)*/                  # zoom or layer number (if torque)
    (-?\d+)/                            # x or z (if torque)
    (-?\d+)                             # y or x (if torque)
    (?:/-?\d+)?                         # y (if torque)
    \.([\w\.]+)                         # extension type
""", re.VERBOSE)


Event = namedtuple("Event", [
    'x', 'y', 'z',
    'time', 'user',
    'ip_address',
    'named_map_template',
    'layergroup',
    'layergroup_timestamp',
    'type'
])


def open_log(path):
    '''
    open_log(string path)

    Opens a compressed (bz2 format) log file.
    '''
    if path.endswith(".bz2"):
        import bz2
        return bz2.BZ2File(path)
    return open(path)
```

```
def tilelog(path):
    '''
    tilelog(string path)

    Reads the log provided by parameters and returns a list of Events. Some
    special lines in the log are ignored.
    '''
    dataset = []
    with open_log(path) as f:
        for line in f:
            if "/static/bbox/" in line or \
                "/static/center" in line or \
                "/favicon.ico" in line or \
                "/attributes/" in line or \
                "/undefined/" in line or \
                "/tiles/layergroup" in line or \
                "GET / MISS" in line or \
                "GET / HIT" in line or \
                "GMT HEAD /" in line or \
                "GET /robots.txt" in line or \
                "GET /https://cartocdn" in line or \
                "GET /rules.abe" in line or \
                "ashbu.cartocdn.com.cartodb.com" in line or \
                "MISS 404" in line:
                continue


            m = LINE_REGEXP.match(line)
            if not m:
                print("Failed to match: " + line +
"-----------------------------------")
                continue


            dataset.append(Event(
                time=datetime.datetime.strptime(m.group(1), "%Y-%m-%dT%H:%M:
%SZ"),
                ip_address=m.group(2),
                user=m.group(3),
                named_map_template=m.group(4),
                layergroup=m.group(5),
                layergroup_timestamp=\
                    datetime.datetime.utcfromtimestamp(
                        float(m.group(6)) / 1000.0),
                z=int(m.group(7 if torque else 8)),
                x=int(m.group(8)),
                y=int(m.group(9)),
                type=m.group(10)
            ))


    print "- - - END OF FILE:" + path + " - - -"
    return dataset
```

12.B. *Python script for tile requests map drawing*

```
def most_requested_events(ds, layergroup):
    '''
    most_requested_events(Event[] ds, string layergroup)

    Given a dataset (list of Events) and a layergroup ID, returns the Z X Y
    values for the most requested events of the specified layergroup ordered
    by frequency.
    '''
    ds = [(d.x, d.y, d.z) for d in ds if d.layergroup == layergroup]
    freq = sorted(((ds.count(d), d) for d in set(ds)), key=lambda x: x[0])
    max_freq = max(f for f, _ in freq)
    return [d for f, d in freq if f == max_freq]


def draw_multimap(tiles, highlight=[]):
    '''
    draw_multimap(Event[] tiles, tiles_dataset[] highlight)

    Given a set of tiles, and assuming a basemap ("basemap.png") colorizes
    the requested tiles according to each level of zoom. The highlight
    parameter allows to emphasize the more requested tiles, which are drawn
    as outlined tiles.
    '''
    basemap = PIL.Image.open("basemap.png").convert("RGBA")
    heatmap = PIL.Image.new('RGBA', basemap.size, (255,255,255,0))
    draw = PIL.ImageDraw.Draw(heatmap)
    w, h = basemap.size
    opacity = 22


    for (zoom, t) in _groupby(tiles, lambda x: x[2]):
        zoomw = (2 ** zoom)
        draw_w = w / zoomw
        draw_h = h / zoomw


        if draw_w == 0 or draw_h == 0:
            break


        for x, y, _ in set(t):
            x = x * draw_w
            y = y * draw_h
            draw.rectangle(
                [x, y, x + draw_w, y + draw_h],
                fill=(255,0,0,opacity)
            )


        for x, y, z in highlight:
            if z == zoom:
                x = x * draw_w
                y = y * draw_h
                draw.rectangle(
```

```
                    [x, y, x + draw_w, y + draw_h],
                    outline=(255,0,0,255)
                )


        opacity += 22


    out = PIL.Image.alpha_composite(basemap, heatmap)
    tmp = tempfile.NamedTemporaryFile(
        suffix='.png', prefix='tmpmap_', dir='.', delete=False)


    with tmp.file as f:
        out.save(f, format="png")


    return tmp.name


def draw_layergroup(events, lg):
    '''
    draw_layergroup(Event[] ds, string layergroup)

    Given a dataset (list of Events) and a layergroup ID, retrieves the
    tiles that correspond to the layergroup ID and draws them in a map. The
    function also computes the most requested events for the layergroup
    which are outlined in the final image.
    '''
    tiles = [(e.x, e.y, e.z) for e in events if e.layergroup == lg]
    highlight = most_requested_events(events, lg)
    return draw_multimap(tiles, highlight)


import urllib2
from io import BytesIO


def build_basemap(zoom, retina=True):
    '''
    build_basemap(int zoom, boolean retina)

    Given a level of zoom, builds a basemap of a valid resolution obtaining
    the needed tiles from an external basemap tile server. If the retina
    option is set as true, tiles are requested with double size.
    '''
    zoomw = (2 ** zoom)
    tile_w = 512 if retina else 256
    w, h = (zoomw * tile_w, zoomw * tile_w)
    basemap = PIL.Image.new('RGBA', (w, h), (255,255,255,255))


    for x in range(zoomw):
        for y in range(zoomw):
            url = “http://3.basemaps.cartocdn.com/light_nolabels/%d/%d/%d
            %s.png"% (zoom, x, y, "@2x" if retina else "")
            print "Requesting %s..." % url
```

```
            raw = urllib2.urlopen(url).read()
            tile = PIL.Image.open(BytesIO(raw)).convert("RGBA")
            tile_w, tile_h = tile.size


            basemap.paste(tile, (tile_w * x, tile_h * y))
            del tile


    print "Writing basemap_%d.png (%dx%d)..." % (zoom, w, h)
basemap.save("basemap_%d.png" % zoom)
```

12.C. *Python script for aggregating and slicing datasets*

```
def get_filepaths(directory, month, day):
    '''
    get_filepaths(string, string, string)

    This function will generate the file names in a directory
    tree by walking the tree either top-down or bottom-up. For each
    directory in the tree rooted at directory top (including top itself),
    it yields a 3-tuple (dirpath, dirnames, filenames).
    '''
    file_paths = []


    for root, directories, files in os.walk(directory):
        for filename in files:
            filepath = os.path.join(root, filename)
            requested = directory + "/ashbu-2015-" + str(month).zfill(2) +
            "-" + str(day).zfill(2)
            if filepath.startswith(requested):
                file_paths.append(filepath)


    return file_paths


def get_sliced_data(file_set, type):
    '''
    get_sliced_data(string[], string)

    This function generates a list of reference/test following an 80/20
    approach from the list of all available files.
    '''
    #By default, training data is set to 80% and test data to 20%
    ref_ = int(round(len(file_set)*0.8))
    if type=="ref":
        print "Number of files in ref set: " + str(ref_) + " out
```

```
            of " + str(len(file_set))
        return file_set[:ref_]
    elif type=="test":
        print "Number of files in test set: " + str(len(file_set)-ref_)
        + " out of " + str(len(file_set))
        return file_set[ref_:]


def aggregate_data_files(file_set, name):
    '''
    aggregate_data_files(string[], string)

    This function obtains a list of files and generates a bz2 compressed
    file based in the concatenation of the files listed.
    '''
    command_ = ' '.join(file_set)
    command_ = "cat " + command_ + " > " + name + ".log.bz2"
    os.system(command_)
    print "The file " + name + ".log.bz2 that contains " +
    str(len(file_set)) + "merged files has been generated"
```

12.D. *Python script to retrieve statistics from log data*

```
def analyze_file_set(file_set):
    '''
    analyze_file_set(string[])

    This function applies an analysis over a list of files.
    '''
    dataset = []
    tileset = []
    files_count = 0
    for f in file_set:
        dataset += tilelog(f)
        files_count += 1
        _write_to_csv('global_analysis', f)
        print "File " + str(files_count) + " out of " + str(len(file_set)) +
        "has been processed"
        tileset = tiles(dataset)
        unique_visitors = set(t.ip_address for t in dataset)
        unique_maps = set(t.layergroup for t in dataset)

        print " ========= STATS ========= “

        _write_to_csv('global_analysis', str(len(dataset)) + " events”)

        print str(len(dataset)) + " events”
```

```
        _write_to_csv('global_analysis', str(len(tileset)) + " tile requests
        (" + str( round( float( len(tileset) ) / float( len(dataset) ) *
        100.0 ,2) ) + "% of total)")
        print str(len(tileset)) + " tile requests (" +
        str( round( float( len(tileset) ) / float( len(dataset) ) * 100.0 ,
        2) ) + "% of total)"
        _write_to_csv('global_analysis', str(len(unique_visitors)) + "
         unique visitors")
        print str(len(unique_visitors)) + " unique visitors"
        _write_to_csv('global_analysis', str(len(unique_maps)) + " unique
        maps")
        print str(len(unique_maps)) + " unique maps"
        _write_to_csv('global_analysis', zoom_stats(dataset))
        print zoom_stats(dataset)
        _write_to_csv('global_analysis', most_requested_layergroup(dataset))
        print most_requested_layergroup(dataset)
        _write_to_csv('global_analysis', " ========= STATS END ========= ")
    return "Done"

def analyze_file(name):
    '''
    analyze_file_set(string[])

    This function applies an analysis over a single file.
    '''
    list_file = []
    list_file.append(name)
    return analyze_file_set(list_file)


def _write_to_csv(name, string):
    '''
    _write_to_csv(string, string)

    This function generates a CSV file or writes to an existing one
    that is referenced through the firs parameter. The string to
    write is passed to the function in the second parameter.
    '''
    with open(name+'.csv', 'a') as csvfile:
        filewriter = csv.writer(csvfile, delimiter=' ',
                                quotechar='|', quoting=csv.QUOTE_MINIMAL)
        filewriter.writerow(string)
```

12.E. *Python script to slice user data chronologically*

```
def analyze_zooms():
    '''
    analyze_zooms()

    This function applies a zoom behaviour analysis over a list of files.
    '''
    fileList = [LIST_OF_FILES]
    dataset = []
    tileset = []
    for f in fileList:
        dataset += tilelog(f)
    tileset = tiles(dataset)
    users = get_users(tileset)
    get_user_activity(tileset, users)


    def get_users(ds):
    '''
    get_users(Event[] ds)

    Given a dataset, returns a unique list of IP addresses that are involved
    in its events.
    '''
    return set(t.ip_address for t in ds)


def get_user_activity(ds, ip_address_set):
    '''
    get_user_activity(Event[] ds, set ip_address)

    Given a dataset (list of Events) and an IP address, returns the request
    activity of the source IP ordered by request timestamp.
    '''
    total_users = 0
    zoom_in = defaultdict(int)
    zoom_out = defaultdict(int)


    for ip_address in ip_address_set:
        events = events_by_ip_addr(ds, ip_address)
        sorted_events = sorted(events, key=lambda x: x.time)


        prev_x = None
        prev_y = None
        prev_z = None
        prev_time = None


        for s in sorted_events:
            prev_time = s.time
```

```
            print "Time: %s X: %d, Y: %d    Z: %d" % (s.time, s.x, s.y, s.z)

            if prev_x is not None:
                total_users += 1
                if s.z > prev_z:
                    print "   Zoom in"
                    zoom_in[prev_z] += 1
                elif s.z < prev_z:
                    print "   Zoom out"
                    zoom_out[prev_z] += 1

                if s.x > prev_x:
                    print "   Pan right"
                elif s.x < prev_x:
                    print "   Pan left"

                if s.y > prev_y:
                    print "   Pan up"
                elif s.y < prev_y:
                    print "   Pan down"

            prev_x = s.x
            prev_y = s.y
            prev_z = s.z

    print zoom_in
    print zoom_out


def events_by_ip_addr(ds, ip_address):
    '''
    events_by_ip_addr(Event[] ds, string ip_address)

    Given a dataset (list of Events) and an IP address, returns the list of
    Events that were requested from the specified source.
    '''
    return [d for d in ds if d.ip_address == ip_address]

def get_random_ip_addr(ds):
    '''
    get_random_ip_addr(Event[] ds)

    Given a dataset, returns one of the IP addresses that requested a
    resource.
    '''
    unique_ip = set(t.ip_address for t in ds)
    return random.choice(list(unique_ip))
```

13. Summary

13.1. Motivation and goals

The main objective of this project is to build an algorithm to predict which map tiles need to be generated and cached whenever a web map is created. This will allow caching these tile resources in advance and avoid their generation on runtime as a result of a user request. In order to achieve this, we will analyse the spatial data contained on thousands of maps together with the interactions that the users perform on them: zooming, moving or clicking on the map features.

13.2. Introduction to web mapping

13.2.1. State of the art

Cartography has been a part of human history for a long time. From the ancient maps of Babylon, Greece or Asia and on into the current century, the different societies that have populated the planet have created and used maps as tools to describe their world.

The improvements in technology and the advent of the Internet have provided a new way to communicate and work which has affected the manner in which maps are built nowadays. In 1996 the first web map was born, but it was 9 years later, in 2005, when Google started to settle the standards that are used in web mapping today. Google Maps introduced then the concept of "tile": PNG images of 256x256 pixels drawn according to the Web Mercator projection that are ordered hierarchically with the zoom level applied to the map. This hierarchical structure is known as a quadtree, a tree in which each node has exactly four children.

Millions of tiles are served each day[1]. In practical terms, being able to predict which tiles of a map will be requested taking into account different parameters such as the characteristics of the underlying data, the kind of represented geometries or their spatial distribution could be the key to build scalable and fast platforms to the map services of the future.

13.2.2. Technical overview of web mapping

In GIS, vector geometries are used to represent the spatial component of geographical features, such as lines, points or polygons. Traditionally, the most common format for storing this geospatial data is the Shapefile[2], which has been used since it was introduced by ESRI in the early 1990s. Modern web maps prefer the KML[3] format, which relies on XML notation, or more recently, GeoJSON[4], based on the JSON format.

To handle geospatial information in a database, PostGIS[5] stands out in the open source world. PostGIS is a module for the open source PostgreSQL database. This module allows PostgreSQL to store geometries in the database and to perform geographical analysis computations.

When a map has to be generated, one of the first concerns lies on how to style this geospatial information. CartoCSS[6] is a language for map design which is similar in syntax to CSS but builds upon it with specific abilities to filter map data. Just like CSS is used over DOM elements, CartoCSS is used over layers. It can also define variables, nest styles or filter the geometries depending on their data attributes.

In the rendering space, Mapnik[7] is the open source engine used to make tile sets. The CartoDB stack that we are using as example in this approach to web mapping uses an intermediate layer above Mapnik: Windshaft[8]. Windshaft is a Node.js map tile server for PostGIS with CartoCSS styling. In order to generate a map, it uses a configuration file containing the information of the data layers that are going to be rendered and their styles. The specification of the layers of a map in these configuration files determines what we know as a *layergroup*. Its identifier is named *layergroup ID*, and it is the key to identify the maps in the system.

Windshaft and Mapnik also take care of the interactivity of the map. When a user clicks or hovers a web map, usually the underlying information is shown as an interactive text box or a popup. In tiled maps, interactivity is obtained from the UTFGrid[9]. The UTFGrid is an invisible tile layer made up of arbitrary letters which are indexes into the clickable data of a map. These are afterwards mapped to

a key from which we can retrieve the rest of data when any interaction occurs within its pixels.

When the different layers of the map are finally generated, a mapping library takes care of displaying them in a website. Leaflet[10] is the leading JavaScript library for interactive maps. Other libraries, such as CartoDB.js[11] or Mapbox.js[12] are built on top of it. These libraries work asynchronously: the tiles and the interactivity information is requested with AJAX calls when required after a user interaction. If the user zooms in, the tiles for the next level of zoom are requested and placed in the correct position in order to build the map.

Further details on any of the processes can be found in *Section 2.2*.

13.2.3. Socioeconomic impact of the solution: regulatory framework

Our tile prediction solution, and the web mapping market in general is not affected directly by any regulatory framework that affects the work of companies or individuals, but there are laws and directives that regulate how local governments offer Open Data and web mapping services. This Spanish law is known as *LISIGE*[13] and regulates the national cartographic system. In Europe, the *INSPIRE*[14] directive was introduced in May 2007.

13.3. Curating and visualizing the source data

In *Section 3* we analyzed the different sources that we will use and how to retrieve and visualize the information that we need from the raw data. Our main source of information are the logs generated from the different CDN nodes.

We also have other two sources for map metadata. The first one is a Redis key-value database that stores the CartoCSS styles that define each map, as well as its interactivity options. The second one is a PostgreSQL database dump, that contains all the information available for a map (such as where is it centered, which is its bounding box, or which level of zoom is used in the initial state of the map).

In order to obtain the relevant information from the logs, we will process them with regular expressions to generate *Event* objects in Python. The *Event* object stores all the information from a request: the type of resource requested (a tile or an interactive UTFgrid request), the XYZ coordinates of the tiles, the time, the source IP address and the map identifier (such as the layergroup and the template name). To handle the information stored in the PostgreSQL tables we will run a query to join the identifier of a map with all the information related to it that lives within the different tables. Then, we will export this result as a JSON file. Similarly, the RDBtools[15] library allows us to export Redis key-value dumps to JSON files.

In the same way that maps are used to represent geographical information, we will use them to have visual information about the tiles that are being requested. The Python functions `draw_basemap`, `draw_multimap`, `draw_layergroup` and `most_requested_events` written for this purpose allow us to visualize the set tiles that are requested as a map. The source code of these functions is available in *Appendix 12.B*.

Log data is divided in several thousand files. To reorganize it, the function `aggregate_data_files` concatenates several files and the function `get_sliced_data` splits the data into two subsets for reference and testing. Their source code is available in *Appendix 12.C*.

In order to have a big picture about the data that we are analyzing, we have run some simple statistics for each of the variables. In terms of events, the total count of them sums up to 1703183, or 58730 events occurring per day. Interactivity requests involve 20% of these requests, while the remaining 80% is due to tile requests.

From a total of 28379 unique maps accesses during the period that we are studying, the most accessed level of zoom is the level 0, being used in the 21,8% of all tile requests. The level of zoom 0 is configured as the one by default in more than 24% of the maps. In terms of location, the mean of the coordinates in which all maps are centered is (-7.9157, 30.5698). We can appreciate a bigger aggregation of maps having their default bounding box at small regions in Europe

and North America, meanwhile less populated places in the world are covered by maps with a wider bounding box.

From the data point of view, the 28379 analyzed maps are formed by 51082 different layers. 53% of these layers are formed by point data, while 22 and 25% are built from lines and polygons.

The Python script to retrieve these statistics from log data can be found in *Appendix 12. D*. The detailed figures and tables in which we visualized the information are available in *Section 3*.

13.4. Designing heuristics for tile caching

The most important variable when it comes to understanding how users will interact with a map is the initial status of the map itself. The initial status of a map is defined by its level of zoom and its center point.

The spatial distribution of the data displayed in a map is the key to define user activity. We note that statistically speaking, users always zoom in on the contents of the map; even if they did not, the process of zooming out would serve mostly tiles which are already cached.

To find the areas on which to focus our caching heuristics, we will calculate an information density ratio for each tile on the map. Tiles which are particularly dense will have a ratio above 1.0 whilst the least populated tiles will be below. This naturally causes our heuristics to emphasize the areas with the most information density.

Unfortunately, and due to constraints on the performance of our tile rendering process, it is not feasible to dynamically trigger caching jobs while the user interacts with the map: our logs on user interactivity show that users spend on average less than 3s on each zoom level of the map. CDN log analysis, however, shows an average of 220ms rendering time spent per tile, allowing us to cache 13 tiles during the average time a user spends in any given zoom level. Given that the default display size of our maps on desktop platforms shows 15 tiles on the screen

at a time, our dynamic caching would always lag behind the actual interaction behavior of the user.

Because of this, our practical implementation will stick to optimistically caching the most likely visited tiles on map generation by running in an offline job on the server side, once and only once after map creation.

We need however to set a hard limit on the amount of time the background job can process and cache tiles, as to not affect the overall performance and availability of our platform. Following existing patterns in our infrastructure, we will limit the total runtime of the background job to 60s in all cases, resulting in the capacity to cache 272 tiles on average.

For our caching algorithm, we will exploit the properties of quadtrees: tile selection will operate as a depth-first traversal of the quadtree, beginning at the initial zoom level of the map, and culling its trajectory based on the data density of each individual tile and the expected zoom level that a user would reach. This simple heuristic, as we will see later on, is trivial to implement and provides great results when applied to real world data.

In practice, the traversal algorithm will not run until a fixed amount of tiles are cached (e.g. 272), but until the timeout for the background job is hit, which means than in some highly dense maps we will not be able to cache as many tiles as on simpler ones.

The depth-first traversal of the quadtree cannot be performed naively, as that would mean focusing all our caching efforts into a narrow view of the map (i.e. one that eventually reaches the maximum zoom level for the first quad we descended into). Because of this, we will cull the depth of the walk at a limit based on the amount of zoom that an user is expected to perform from a given zoom level. This culling however will be biased based on the expected amount of information density in that branch of the quadtree.

The caching traversal begins at the initial set of quads for the map and will descend as many zoom levels as the average values seen in Table 16. For each

level of recursion, we will decrease the depth accordingly but we will also adjust it based on the information density ratio for the quad we are descending into. This way we make sure that the most dense quads will be cached more frequently than their simpler counterparts.

To verify our heuristics, we will tweak our caching algorithm as to *mark* the cached tiles on a blank basemap. With this, we will be able to compare the interaction patterns with the renderings from our caching heuristic and verify visually that the cached tiles are an useful and accurate subset of the actual interactions performed by our users.

13.5. Real applications of cache generation

In order to study the impact of cache generation in a real scenario, we are going to calculate the time which is taken to generate different kinds of map tiles. The first step of the tile rendering process is to obtain the data that needs to be shown in the tile through SQL. We will refer to the time of this operation as T_{DATA}. Windshaft uses several background threads to process rendering requests. The amount of time that the process is queued waiting for a thread will be defined as T_{QUEUE}. Finally, when the actual rendering process has started, the tiler spends a variable amount of time generating the tiles, which depends on geometry complexity and on the styles applied to the features. This amount of time will be defined as $T_{RENDERING}$. Therefore, the time needed for rendering a tile (T_{TILE}) can be defined as the sum of T_{DATA}, T_{QUEUE} and $T_{RENDERING}$.

In *Section 5*, we performed some measurements for different tile scenarios, such as tiles with different data aggregation and tiles generated from advanced and simple queries or styles. These different experiments showed differences of time from 124ms up to 11 seconds with respect to serving cached tiles.

13.6. Project planning and budget

The project is divided in four main stages: the obtention and curation of the data, its analysis, the study of different statistical approaches and the application of

these to provide a final predicting model over the source data. From these four stages, we can define 12 activities with a total duration of 297 hours — or 75 days estimating an amount of 4 hours of work per day. The temporal planning together with the Gantt and PERT diagrams, as well as further details on the tools to be used along the project can be found in *Sections 6.1.3. and 6.1.4.* respectively.

In terms of budget, caching our resources in a content delivery network network implies paying the cost of this service. In *Section 6.2.* we analyze the pricing of three content delivery companies: MaxCDN, Akamai and Fastly, which charge per bandwidth and per number of HTTP and HTTPS requests received. MaxCDN appears to be the most competitive option in terms of cost per served tile. It has a cost of $0,0019 per 1000 tiles, while Akamai and Fastly charge $0,014 and $0,0134, respectively.

13.7. Conclusions

Throughout this document we have studied the most important concepts of web mapping as an introduction to the most technical details involved in tile generation. We have curated hundreds of gigabytes of data, with the goal of using it to build a predictive system for the tiles that are likely to be requested depending on their geospatial properties. After testing the generated algorithm to prove its efficacy, the outcomes are optimistic. Despite the fact that extremely dense maps will not be completely cached because of the big amount of tiles implied in them at high resolutions, the regions that contain a lot of information will be covered. This is done thanks to the weigh that they imply in our caching algorithm, which will go through the most populated children of the quadtree in contrast with the less populated ones, that will be skipped.

By performing an analysis on tile rendering times, we have confirmed that this solution improves the response time of the maps at the same time that it allows us saving rendering resources.

13.8. Terminology (summary)

***AJAX* (Asynchronous JavaScript and XML).** Techniques used on the client-side of web applications that allow sending and retrieving data from a server without interfering with the display and behaviour of the existing page

CartoCSS. Language for map design developed by Mapbox, similar in syntax to CSS but oriented to styling geometries.

CDN (Content Delivery Network). Network of distributed servers that deliver web content to a user based on their geographic location.

GeoJSON. Format for encoding geographic data structures based on the JavaScript Object Notation (JSON) format.

GIS. Any system designed to capture, store, manipulate, analyze or present all types of spatial or geographical data.

KML. File format used to display geographic data in a mapping application based on the XML format.

Mapnik. Open source mapping toolkit for tile rendering, written in C++.

PostGIS. Open source extension for PostgreSQL that adds support for spatial functions and geometry data types to the database.

PostgreSQL. Open source object-relational database system.

Redis. Open source and BSD licensed advanced key-value cache and store.

Shapefile. Vector data storage format for storing the location, geometry, and attributes of geographic features.

Tile. Image of 256x256 pixels that represents a fixed geographical region and makes up a map once combined with adjacent tiles of the same level of zoom.

UTFGrid. Invisible tile layer made up of indexes and data that is shown in the interactive options of a map.

Web Mercator (projection). Variation of the Mercator projection which is the de facto standard for Web mapping applications.

Windshaft. Map tile server written in Node.js for PostGIS with CartoCSS styling.

13.9. References (summary)

[1] CartoDB, Inc. *The versatility of retrieving and rendering geospatial data with CartoDB*. May 16, 2013. http://blog.cartodb.com/the-versatility-of-retreiving-and-rendering-geospatial/

[2] Wikipedia contributors. *Shapefile format*. Wikipedia, The Free Encyclopedia https://en.wikipedia.org/wiki/Shapefile

[3] Wikipedia contributors. *Keyhole Markup Language format*. Wikipedia, The Free Encyclopedia https://en.wikipedia.org/wiki/Keyhole_Markup_Language

[4] Butler, Howard; Daly, Martin; Doyle, Allan; Gillies, Sean; Schaub, Tim; Schmidt, Christopher. *GeoJSON format*. http://geojson.org/geojson-spec.html

[5] *PostGIS*. http://postgis.net/

[6] *CartoCSS*. https://github.com/mapbox/carto

[7] *Mapnik*. http://mapnik.org/

[8] *Windshaft*. https://github.com/CartoDB/Windshaft

[9] *UTFGrid specification*. https://github.com/mapbox/utfgrid-spec

[10] *Leaflet*. http://leafletjs.com/

[11] *CartoDB.js*. https://github.com/CartoDB/CartoDB.js

[12] *Mapbox.js*. https://www.mapbox.com/mapbox.js

[13] Spain. *Ley 14/2010 sobre las infraestructuras y los servicios de información geográfica en España* . http://www.idee.es/web/guest/espanol-lisige

[14] European Union. *Directive 2007/2/EC of the European Parliament and of the Council of 14 March 2007 establishing an Infrastructure for Spatial Information in the European Community*. http://inspire.ec.europa.eu/

[15] *RDBtools*. https://github.com/sripathikrishnan/redis-rdb-tools