High performance GeoSpatial application, using Twitter to increase trace accuracy

Project for Ericsson

Simon Schneider
Abstract

High performance GeoSpatial application, using Twitter to increase trace accuracy

Simon Schneider

The goal of the project is to research new technologies and ways to create high speed and high performance applications in order to deal with GeoSpatial data. The research data set used is GEO-located Tweets. Twitter is chosen for its high data amount, accessibility and because of its relevance to cellular network optimization. Beyond the research part of the project, the end product will be used to increase cellular trace accuracy through the use of the highly accurate GEO-located-Tweets.

The result of the final product is showing great potential. An interactive web map rendering 700 million Tweets up to a 10 by 10 meter resolution within seconds, while also allowing the user to pan and navigate the map without any performance issues.

The scope of the project includes the entire process, from aggregating Tweets with a Java program, storing them in a PostgreSQL+PostGIS database, creating maps and GIS-files with MapServer, to presenting Maps and serving downloadable files in a Django web application utilizing OpenLayers.

The used procedures are explained and justified in great detail throughout this report. All products used in the project are licensed under the open source initiative and are therefore free to use by anyone.

The report also includes suggestions, relevant products, research projects and algorithms for future development and research.
Contents

1. Background and Purpose 5

2. Method 5

3. Collection of Data 6
   3.1. Twitter Streaming API & Twitter4j 6
   3.2. Producer-Consumer-Design 7
   3.3. Database batch-insert 7

4. Storage of Data 7
   4.1. PostgreSQL+PostGIS 8
   4.1.1. PostgreSQL vs MySQL 8
   4.2. Tables 8
   4.2.1. Cache 9
   4.2.2. Raw Data 9
   4.2.3. Binned Data 10
   4.3. Design Choices 10
   4.3.1. Indexing 10
   4.3.2. Flush Event 11

5. Server side processing 11
   5.1. MapServer 12
   5.1.1. Maple 13
   5.1.2. Layers 15
   5.2. GRD-Creation Program (Java) 15
   5.3. GeoSpatial information 16
   5.3.1. Projections 16
   5.3.2. Raster vs. Points 17

6. Client side web application 17
   6.1. Django 17
   6.1.1. Models 19
   6.1.2. Views 19
   6.1.3. Templates 19
   6.1.4. Static files 20
   6.2. OpenLayers 20

7. Conclusion 20

8. Future development and Research 20
   8.1. MapD 21
   8.2. MongoDB 21
   8.3. ClusterHulls 21
   8.4. Storm 22

References 22
1. BACKGROUND AND PURPOSE

Part of mobile network optimization relies on location based information of users. A network optimization engineer will evaluate this data and make changes to the network accordingly. As the data sets used for these decisions are very large it is a significant challenge to design software systems to deal with them. The data has to be processed and drawn onto maps as fast and reliably as possible to provide the optimization engineer with tools that do not limit his/her efficiency.

The purpose of the project is to investigates new tools, programs and designs in order to battle the problems of large GeoSpatial data.

The testing dataset used for the investigation comes from Twitter. The data is used, not only because of its high accuracy and substantial size, but also because it will be used as one of the data sources for optimization. The high accuracy and ability to pick up users outside mobile networks, i.e. WiFi, makes it a beneficial dataset in the process of optimization.

2. METHOD

The design of the project can be divided into 4 distinct parts, the collection of data, the storage of it, the server side processing and the client side visualization.

Each of them are a separate stand alone project that can easily be replaced or with minor adjustments work with other technologies. The individual components interface with each other through network, allowing, but not requiring, the components to be deployed on separate physical machines. A visualization of the design can be seen in Figure 1.

Collection

The collection of data is managed by a multi threaded Java program, it collects Tweets via the Twitter API and sends them to the Storage in batches, detailed explanation in Section 3.

Storage

PostgreSQL is the chosen database engine for the storage component of the application, PostgreSQL is also extended through the popular spatial extension PostGIS. The design and procedures used for the database are explained in Section 4.

Server side processing

MapServer is used as a WMTS, it manages the server side processing of the data in the database. When a web request is relayed to MapServer it collects the relevant information from the database and creates images, map tiles, accordingly which are then sent as a response to the request. A complete explanation in Section 5.

Client side visualization

The client side consists of a web application built in Django, a python based web framework. It utilizes the OpenLayers JavaScript to overlay the map tiles from MapServer onto world maps from Google Maps and OpenStreetMaps. Complete documentation is found in Section 6. The client side consists of a web application built in Django, a python based web framework. It utilizes the OpenLayers JavaScript to overlay the map tiles from MapServer onto world maps from Google and OpenStreetMaps. Complete documentation is found in Section 6.
An Apache web server is used to host the Server side applications, but will not be discussed individually as configuring it is considered part of basic setup.

3. COLLECTION OF DATA

The collection of Tweets from Twitter is managed by a multi threaded Java program, designed after a classic Produce-Consumer Problem (see Section 3.2), connecting to the Twitter Streaming API (see Section 3.1). The program sends the filtered and stripped Tweets to the database in batches (see Section 3.3).

3.1. Twitter Streaming API & Twitter4j

The open Twitter API gives access to a variety of information to be requested from the Twitter Server. The primary Tweet information used in this project is the geographical location, the ID, the user ID of the owner and the time of creation of a Tweet.

Simplified the Twitter API works with incoming queries, requests for certain Tweets, which the server responds to with a JSON object containing the requested information.

The Twitter Streaming API and Twitter Rest API are the two main components of the Twitter API. In this project the Streaming API is used.

The Streaming API lets a client set up a private streaming connection to the Twitter server requesting Tweets conforming to a certain filter. The server will send the conforming Tweets in an endless stream to the client.

The free license of the Twitter API, available to any Twitter user, has certain limitations. The Streaming API limits any connection to receive a maximum of 1% of all Tweets at any given time [10]. In this project the data is filtered to only GEO-tagged Tweets which limits the amount to about 10%, which is still too high. To circumvent this, multiple accounts are used, collecting data from different areas of the world, effectively nullifying the restriction and allowing access to all the relevant Tweets. More information about the Twitter API can be found on the official documentation [8].

Twitter4j is a Java library created to streamline the integration of the Twitter API into Java. It eases the querying to the server and parses the incoming JSON objects into Java Objects. This project exclusively uses Twitter4j to interact with the Twitter API.

More information about Twitter4j can be found on the official documentation [7].
3.2. Producer-Consumer-Design

The producer-consumer-problem is a well known basic programming problem [3] dealing with a producer that creates data in an endless stream and a consumer that is supposed to process this data.

The solution to the problem is a multi threaded design utilizing a queue [3]. The producer thread writes data into a queue while the consumer thread reads and processes the data from the queue. The multi threading is important because all threads can work independently of each other, the queue works as a buffer between them allowing them to work at different speeds. The consumer still has to work at an equal or higher average speed than the producer though. If this is not the case, the issue of an ever growing queue can be solved by adding multiple consumer threads.

This project uses the producer-consumer-design outlined above, with multiple Producer threads. One thread per Twitter account, each one sets up a Twitter streaming connection and collects Tweets for a certain part of the world (more information see Section 3.1). The collected Tweets are put into a LinkedBlockingQueue holding Tweet Objects that consist of Tweet ID, User ID, Time-stamp and Location.

The Consumer consists of a single thread, multiple threads are not needed in this design as the work load of the Consumer is rather low. The consumer will continuously check if the queue contains more than a specified number of Tweets. If this is the case it will open a connection to the database and send a batch-insert-command. Explanation why batch inserts are used in Section 3.3.

3.3. Database batch-insert

The insert into the database is done in batches for a couple of reasons.

First and foremost it is done to lower the amount of interactions with the database. Instead of having a constantly active consumer thread taking a lot of processing power, the thread can insert multiple Tweets at once and then sleep for a few seconds.

Secondly, to insert in batches also opens the option to close the connection to the database between inserts. Even though it takes some resources to open new connections, the benefit of having an extra, mostly open connection, is better for the database.

Thirdly because PostgreSQL is a transactional database it is very beneficial inserting large quantities of data in a single statement, batch insert. The reason is that the database will internally create versioning and roll back procedures for every single insert statement or transaction. This creates an immense overhead when inserting large quantities of rows in individual statements and also wastes disk space.

4. STORAGE OF DATA

The database engine used in this project is PostgreSQL, it is one of the most advanced open source database engines available and extended through PostGIS. PostgreSQL+PostGIS is one of the leading open source spatial database engines, detailed explanation on PostgreSQL+PostGIS in Section 4.1.

The database is designed with a cache which stores the last couple of hours of data. The database has a raw data table containing the full amount of data and also multiple binned-
data tables containing preprocessed data. Detailed explanation is given in Section 4.2. Part of the data in the cache is periodically flushed into the raw data and binned-data tables. That and other design choices are explained in Section 4.3.

4.1. PostgreSQL+PostGIS

PostGIS is the spatial extension for PostgreSQL which adds functionality for creating geographical Points, Rasters, Lines, Polygons etc.. Further PostGIS also includes various functions to interact with and manipulate these geometries. PostGIS is also projection aware and will reproject geometries between various projections.

The geometry type used in the Project is of Point-type which is stored in projection EPSG:4326 and EPSG:3857, more on projections and their relevance is given in Section 5.3.1. More information about why Points are chosen as the storage type is give in Section 5.3.2. The points are stored as WKT in a geometry column which is the standard for representing geometry in programming.

4.1.1. PostgreSQL vs MySQL

In this project the choice of open source database engine is between PostgreSQL and MySQL. MySQL has for a long time been regarded faster [1] and easier to use while PostgreSQL has been known for being more robust and complete, but harder to work with [2]. MySQL employs fast read tables, so called MyISAM, with table wide write locks. MyISAM tables have high-speed reading but as mentioned very limited write possibilities. These tables have long been the decisive factor in favor for MySQL when speed has been compared between the two database engines. PostgreSQL does not feature any similar table mechanics.

In the last releases of PostgreSQL the speed of the database engine has largely been catching up to the speed of MySQLs MyISAM tables without sacrificing multiple writes and versioning.

Moreover, the most critical part of the project is the database engines spatial implementation. In this regard MySQLs spatial implementation pales in comparison to that of PostgreSQL which features abilities close to that of OracleSpatial. The speed difference between MySQL and PostgreSQL when working with spatial data is also in favour of PostgreSQL because of its better spatial indexing with GiST indexes. To conclude, for this project PostgreSQL+PostGIS is the most appropriate database engine. The reason for that is its extensive spatial implementation, superior indexing as well as its high speed and feature richness.

4.2. Tables

The main tables in the database are a cache (Section 4.2.1) which stores the last couple of hours of data, a raw data table (Section 4.2.2) containing the full amount of data and multiple binned-data tables (Section 4.2.3) containing preprocessed data.
4.2.1. Cache

The Cache main task is to be a buffer between the collector (Section 3) and the storage (Section 4.2.2), there are a two reasons for this.

1. The Twitter API will frequently send a single Tweet multiple times, therefore there is a need for a process that catches this issue. The solution is a short term cache storage on which a SELECT DISTINCT statement can be used to insert into the raw data. More on this flush event”in Section 4.3.2.

2. Because of the high number of inserts into the table from the Collector, even considering batch inserts (Section 3.3), it is beneficial having a small table that the data is being inserted into. This because every insert will create a snapshot of the table for roll back reasons, therefore it is advised to only insert with high frequency into small tables.

For these reasons a cache is introduced, the cache in the project contains the three most recent hours of Tweets. Table I shows the column layout for the cache data table.

<table>
<thead>
<tr>
<th>Tweet_ID</th>
<th>BIGINT</th>
<th>NOT NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>User_ID</td>
<td>BIGINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Tweet_Timestamp</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Tweet_Location</td>
<td>GEOMETRY</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>

Table I: Column layout for Cache table.

The geometry points in the cache are in projection EPSG:4326, more on projections in Section 5.3.1.

4.2.2. Raw Data

The raw data table contains all the collected data and serves mainly as a backup of the collected data. It is also used for the GRD writer program, see Section 5.2. The geometry points in the raw data are in projection EPSG:4326, more on projections in Section 5.3.1. Table II shows the column layout for the raw data table.

<table>
<thead>
<tr>
<th>Tweet_ID</th>
<th>BIGINT</th>
<th>NOT NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>User_ID</td>
<td>BIGINT</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Tweet_Timestamp</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>Tweet_Location</td>
<td>GEOMETRY</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>

Table II: Column layout for Raw Data table.
4.2.3. Binned Data

The binned data tables are the main tables being used for the web application, there is one binned data table for each zoom level of the map. The geometry in these tables are reprojected to EPSG:3857, the Google Pseudo Mercator projection, before the binning. The binning formula is as follows:

\[
\text{Mapextent} = 20026376.39 \\
\text{Scale} = \frac{\text{Maps}ize}{256 \times 2^{\text{zoom}-1}} \\
\text{Pixel\_location} = \left(\left\lfloor \frac{\text{Point\_location} + \text{Mapextent}}{\text{scale}} \right\rfloor \times \text{scale} \right) - \text{Mapextent}
\]

The data is then grouped by the pixel\_location geometry and the number of points in each Pixel\_Location are counted in number\_of\_tweets. For the table there is also a pixel\_ID, a serial counter, attached to each pixel because of MapServer requirements. Table III shows the column layout for the binned data tables.

<table>
<thead>
<tr>
<th>pixel_ID</th>
<th>SERIAL</th>
<th>NOT NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>pixel_Location</td>
<td>GEOMETRY</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>number_of_tweets</td>
<td>INT</td>
<td>NOT NULL</td>
</tr>
</tbody>
</table>

Table III: Column layout for binned data tables.

These binned-data tables will be queried by MapServer, more about the MapServer layers and querying in Section 5.1.

4.3. Design Choices

There are some underlying designs to consider when working with the database, Indexing (Section 4.3.1) and Events+Procedures for flushing the cache (Section 4.3.2).

4.3.1. Indexing

An index is an internal data structure that improves data retrieval speed. PostgreSQL offers various indexing mechanics, B-Tree indexes for normal columns and GiST indexes for geometry columns are the ones used in this Project. PostgreSQL also offers a Cluster mechanic which allows the physical reordering of data on the hard drive in order to conform to the index order. This increases the lookup speed even further as not just the index, but also the actual data is correctly ordered on the disk. The use of an index increases query performance but also has some drawbacks. An index requires additional disk space and also decreases the write performance of the table. In addition to this some indexes may even decrease query performance as they can "confuse" the query-planner and therefore lead to counter productive choices by the planner. The following indexes are used in the corresponding tables.
Raw Data table, see Section 4.2.2

This table contains two indexes, one B-Tree index on the tweet_timestamp column and one GiST index on the tweet_location. Because the amount of data in this table constantly increases the data is not clustered on any of the indexes.

Binned Data table, see Section 4.2.1

These tables contain two indexes each, one B-Tree index on the pixel_ID and a GiST index on the pixel_Location. Because of the rather stable rows in these tables and the high performance requirements, the tables are clustered on the GiST index.

4.3.2. Flush Event

The event of flushing the cache is a procedure executed every 1 hour, it inserts distinct rows from the cache into the raw data table while also updating or inserting the relevant pixels in the binned data tables. It is only elements of the cache older than 3 hours that are selected by the procedure and after the insert the rows are deleted from the cache.

The procedure is performance heavy and poses quite a challenge to optimize to a reasonable execution time. There are several performance pitfalls that need to be addressed, most of them are related to the update of binned data tables. The data has to be binned to every single zoom level, already existing pixels have to be updated accordingly and other pixels have to be created.

The following procedure is used to circumvent these issues, the procedure is written in python to allow parallelizing of calculations.

1. A main temporary table is created and distinct old data from the cache is inserted 1 to 1 into it.

2. The following steps are run in parallel.

   (a) The data is copied from the main temporary table into the binned data table.

   (b) The following three steps are run simultaneously for each binned data table

      i. A binned temporary table is created for every binned data table.

      ii. The data from the main temporary table is reprojected to EPSG:3857 and binned into the binned temporary tables.

      iii. The binned temporary tables are used to insert and update the binned data tables.

3. The old data from the cache is deleted and the temporary tables are dropped.

This design utilizes the maximum amount of parallelizing in order to execute the event with an acceptable execution time.

5. SERVER SIDE PROCESSING

   Server side processing and WMTS have three key advantages compared to client side processing.
Speed

Speed is the main advantage as the server will handle all heavy calculations and processing of data. The server will always have more RAM, faster disks and a faster processor than the client where the application is used. Furthermore, the server will or should be connected to a high speed internet connection which results in higher transfer speed between database and WMTS. On top of this the server will be optimized for one task and not have any unnecessary programs installed.

Scalability and Security

Another important advantage of using a WMTS is that the WMTS, not the client, will be connecting to the database. This is both a security and scalability advantage.

Foremost the client will never see or receive any information about the database connection details, it will also never receive the actual data but instead only the picture representing the data. This is a security and bandwidth advantage.

Secondly, the connections to the database will not increase as the number of users increases. This because the WMTS will always have a limited number of connections to the database regardless of the number of users. However, because of caching of image tiles this does not have an adverse effect on performance as most tiles will be cached when dealing with a lot of users.

The server side processing of data and serving of maps is managed mainly by MapServer which is set up as a WMTS. It reads data from the database, creates image files accordingly and sends them on to the web application. A simplified illustration can be seen in Figure 2.

The other processing of data is a client GRD-writer which is written in Java, it reads the raw data table and writes the results into a GRD file that is used in TPS, more on the GRD-write in Section 5.2.

5.1. MapServer

*MapServer is an open source development environment for building spatially enabled internet applications. It can run as a CGI program or [...] [5]*

This project uses Apache web server where MapServer is run as a FastCGI program and is set up as a WMTS.

The way the WMTS service works can simplified be seen as follows.

1. Apache will receive a tile request, through an URL, that references the FCGI MapServer program.
Apache will call MapServer and pass any text after the `/mapserv.exe?` as parameters to the call.

2. When MapServer receives the WMTS request it will parse the parameters. These parameters, among other things, contain references to the following:

**Map file**
- A reference to a Map file which is a MapServer specific file containing the MapServer code. More information about the Map file in Section 5.1.1

**Layers**
- The name of the layers to be drawn, these layers have to be referenced in the Map file.

**BBox**
- A bounding box, defining the requested area.

**Width and height**
- Defining how large, pixel wise, the requested tile should be.

**Options**
- Various other options defining how to draw, style and/or format the requested tile

3. MapServer will parse the defined Map file for the requested layer. In the layer a table from the database is defined as the data source. This table will be queried for the number of Tweets of each pixel contained in the bounding box.

4. The result from the above query will be drawn onto a png image where each pixel is coloured according to the number of Tweets in that pixel. Note, to achieve a heat-map effect the number of tweets will coloured on a gradient from blue to red.

5. The image is returned to Apache which will send it as a response to the client.

The reason why FastCGI is used opposed to regular CGI is because FastCGI does not close executed programs right away but keeps them alive for a certain time frame after a completed request. For MapServer this is a performance gain as it allows connections to the database to remain open and this removes the main overhead from the interaction with the database.

The programming interface of MapServer, the Map files, will be shallowly explained in Section 5.1.1. For a complete documentation of Map files see the official MapServer Documentation [4].

### 5.1.1. Map file

The Map file is the programming interface of MapServer, it is a .map file that is passed as a parameter to the mapserv.exe program.

A very basic structure of a Map file is shown in Code 1.
Co de 1: Sample Map file (note that this is only an incomplete non working example)

MAP

NAME "NameOfMapFile"

PROJECTION

"init=EPSG:3857"

END

UNITS meters

WEB

//Web Information

METADATA

"wms_title" "NameOfMapFile"

//other WMS information

END

END

LAYER

GROUP twitterautobgr

TYPE POINT

MAXSCALEDENOM 70000000 //Max Scale of the Layer

MINSCALEDENOM 40000000 //Min Scale of the Layer

SYMBOLSCALEDENOM 55467893 //Scale of the Symbols in the Layer

CONNECT "database connection details"

PROCESSING "CLOSE_CONNECTION=DEFER" //FastCGI performance option

DATA "pixel_location FROM binneddata_03 using unique pixel_id using srid=3857" //Basic select statement

CLASSITEM "number_of_tweets" //the column which we want to filter by when coloring

CLASS //Defines display options (filters according to classitem)

EXPRESSION ([number_of_tweets] >= 10 and [number_of_tweets] < 1500)

STYLE

SYMBOL 'square'

SIZE 1

COLOR 0 0 255

END

END

//More Classes Defining coloring of other number_of_tweets

SYMBOL //Defines the Symbol to be drawn.

NAME "square"

TYPE vector

POINTS

0 0

0 1

1 1

1 0

0 0

END
There are several options not shown in the sample code, but it gives a rough outline of the Map file.

- The Map Section defines default parameters like output projection, unit of measurement as well as default web and WMS metadata.
- The Layer Sections define specific Data parameters like a database table, colouring according to a number count, symbol type and size to be drawn and various other things.
- The Symbol Sections define Shapes that can be drawn onto the map i.e. Points, Lines, Squares.

More information about the Map file in the official MapServer Documentation [4].

5.1.2. Layers

This project uses point type layers exclusively, they are used as makeshift rasters, see Section 5.3.2. This project utilizes layers to query the various preprocessed zoom level tables, see Section 4.2.3. The reason for using one table per zoom level is that at any given zoom level, any given tile can request a maximum of $x_{pixel} \times y_{pixel}$ number of points independent of zoom level. This is an excellent solution to keep transfer and query times of the database to a minimum. There are two categories of layers used, auto-switching and specific.

**auto-switching**

These layers utilize the maximum and minimum scale option in MapServer to automatically switch between each other depending on the requested zoom level (scale level). The different layers will query the appropriate table in the database. This results in a smooth and fast zooming and panning experience for the client. The reason is that all the calculation related to switching layers is done on the server, not in the client JavaScript.

**specific**

These layers let the user select any layer at any zoom level even if the zoom levels do not match. This is at expense of a minor performance loss though as the client side JavaScript has to do more work, the performance loss is hardly noticeable though.

5.2. GRD-Creation Program (Java)

Right now the Ericsson tool that will use Twitter Data for network optimization is the TPS tool "ECO-project". This tool takes .grd files as input, therefore a Java based GRD-creation-application is developed.
GRD files are binary GIS files that are comparable to pictures. They include header which contains GIS-information about the data contained. They also include the actual data which is a raster with a double precision number in every cell.

The GRD-creation-application built in this project will take a GRD-file as input and create a new GRD-file containing Twitter Data. The reason why a GRD-file is used as input is that an error free header has to be read and copied. To let users enter the header information manually was ruled out because of high risk of a user error. Errors could result in corrupted files or in the worst case produce working files containing misleading information.

The GRD-creation-application will read the header of the input file and query the Twitter database accordingly. The result of the query is written into a new GRD-file that is saved in the same folder as the original file. The new GRD-file can be imported to TPS/ECO-project as a clutter.

A sample of the GRD-write interface is shown in Figure 3.

![Grd2TwitterGrid](Image)

Figur 3: The GRD-writer Interface

5.3. GeoSpatial information

5.3.1. Projections

Projections play a big role in GeoSpatial analysis, they define how the elliptical world is projected onto a two dimensional plane as well as how to reference a point on this plane. This project uses two projections, EPSG:4326 which is unaltered latitude longitude and EPSG:3857 a spherical pseudo Mercator projection developed by Google.

**EPSG:4326**

This is not really a projection but simply unaltered latitude longitude expressed in radians. EPSG:4326 is useful for storing information as it is the common input parameter for almost any GIS application. But all applications will internally reproject them before showing them on a map because this projection will not work if the map has a high resolution. The biggest issue is that it will distort any angles, i.e. streets will not meet perpendicular.

Therefore it is mainly used as a storage projection in this project.

**EPSG:3857**

This is a Mercator projection developed by Google and is used in mostly any web
application. It relies on that the map should mainly be viewed at a street or city level. It distorts the high level map, i.e. the closer to the pole the more stretched the map is. This is because in EPSG:3857, meridians and parallels will meet at a right angle, this will ensure that angles are correct but will distort maps to infinity at the poles. Google argues that angles are most important as maps must conform to what people see at ground level, not that maps are perfectly proportional at a high scale.

The unit of measurement in EPSG:3857 is meters.

In this project EPSG:3857 is used for the web application and the preprocessed data.

5.3.2. Raster vs. Points

The data that is displayed in this project should be displayed as a raster, the world is divided into equal size squares with a number of Tweets in each. The issue with storage as raster is that all raster storage engines require every single point to have a number defined, even if it is 0 or NULL. Because of the resolution requirements, 10m * 10m, this is impossible as this would require about 6TB of disk space for only the earth’s land mass. The solution is instead to only save the points with an actual value, the points are then displayed as squares and will therefore achieve the same effect as a raster.

6. CLIENT SIDE WEB APPLICATION

The client side web application is built in Django, it utilizes OpenLayers to display the map and take care of WMTS requests as well as various other JavaScripts for the interface.

The interface offers a large map and a menu which lets the user select a base layer as well as multiple overlays. The menu offers categories for structuring the layers, the base layer category is mandatory while others can be added and removed manually. The adding of base layers, layers and categories is done through an administrator portal. This administrator portal streamlines the process of adding new layers and editing the web application as it is not necessary to do any code alterations for simple changes. A sample of the web application interface is shown in Figure 4.

6.1. Django

Django is a high level python based web framework, it features among other things database integration, extensive automation permitting rapid development as well as an integrated administration system.

In this project Django is built with a PostgreSQL database for managing the web application. Note that this is a separate database from the Twitter one.

Any Django web application is built on 4 corner stones which are the Models, the Views, Templates and static files. Models are the data, Views are the logic behind the dynamic interface and application while the templates define the look of the interface. Static files are additional files needed for the web page, i.e. CSS, JavaScript and images. A quick introduction to each cornerstone is given below.
Models
Models are the objects or sources of data, in Django. They reference tables in the database in which the instances of the object is saved. Each field in an object is a column of the table and each instance of an object is a row in the table.

Models are written in Python.
The models used in this application are described in Section 6.1.1.

Views
A view in Django is the heart of a single web page, it responds to requests with the actual html file. These html files are created on the fly according to the data in the objects. Views are connected to a single template that defines the dynamic html file.

Views are written in Python.
The view used in this application is described in Section 6.1.2.

Templates
A template is a html document with Django specific syntax that allows a programming approach to creating html. Among other things it adds functionality for loops, statements and object access when writing the html. Once a view receives a request it will parse the template into a real"html document. Note that an html document also can include in-line JavaScript or CSS, therefore it is possible to have dynamic JavaScript and CSS as well.

Templates are written in html with Django specific syntax to create dynamic content.
The template used in this application is described in Section 6.1.3.

Static Files
As the name suggests static files include any files that are static and constant, i.e. images, CSS or JavaScripts. They are sent to the client unaltered through the web server when requested.
The static files used in this application are described in Section 6.1.4.

The administration portal of Django streamlines the addition of new- or editing of old-Models. As a result of this it is very easy to manipulate and edit the application without touching any code.

More information about Django can be found on the official documentation [6].

6.1.1. Models

The models used in this application are the following:

Base Layers
This model represents a base layers of the application, it has two parameters, a name and an addtext". The addtextparameter is an OpenLayers new layer argument written in JavaScript.

The base layers used are google maps and open street maps base layers.

Layer Categories
This model represents a category used for structuring overlays, it only has one parameter, the name.

The categories defined are auto adjusting- and specific- layers.

Overlays
This model represents an overlay of the application, it has three parameters, a name, a Layer category and an addtext". The addtextparameter is an OpenLayers new layer argument written in JavaScript.

The overlays used are the auto adjusting as well as the specific versions of the Twitter MapServer WMTS.

6.1.2. Views

There is a single view in the application which is the map view. A web page showing the map from OpenLayers and a menu containing the base layers and overlays.

The template connected is described in Section 6.1.3.

6.1.3. Templates

Apart from the administrator template used for the administration portal, which will not be described in detail, there is one template used in this application. It shows a map from OpenLayers as well as a menu containing the base layers and overlays.

The template includes the dynamic creation of html elements for the base layers and overlays. It also features in line JavaScript to add these base layers and overlays to OpenLayers and attach listeners related to turning them on and off. This in line JavaScript acts as a dynamic extension to the application specific JavaScript included in the static files.

19
6.1.4. Static files

Apart from an application specific JavaScript the static files served for this application include OpenLayers, JQUERY, JQueryUI as well as various CSS://: for styling.

The application specific JavaScript creates the OpenLayers Map and manages among other things the base layers, overlays and event listeners. It is extended by an in line JavaScript in the template to manage the dynamic addition of these things according to Django Models.

6.2. OpenLayers

OpenLayers is a JavaScript library for displaying maps on web pages. It is an open source alternative to Google Maps and Bing Maps.

The important features for this project is the support for WMTS, it overlays the tiles from MapServer onto any base layer. It also supports many of the big base map providers such as Google and OpenStreetMaps natively which eases the incorporation of these.

In addition it also supports all other major features expected from a web mapping service which might be interesting for the future of the project.

7. CONCLUSION

The result of the final product is showing great potential. An interactive web map rendering 700 million Tweets up to a 10 by 10 meters resolution within seconds, while also allowing the user to pan and navigate the map without any performance issues. The application and tools developed within the project show the great potential of the technologies and designs used. It is also a valuable source of knowledge that Ericsson will use for future development. Ericsson has already received the tools and they are now part of an in house release of TPS. After some iterations and market research the tools will also be included in the customer versions of the software.

The fact that all software and products used in the project are licensed under the open source initiative is an advantage to any business. Not only as it is free but also because the business is not reliant on other companies that own the source code of the software.

The application can right now be found on a in house Ericsson server [14], the source code is also available on a Ericsson repository [15].

8. FUTURE DEVELOPMENT AND RESEARCH

Possibilities for further development can be divided into two parts.

Firstly it is possible to add other external data sources like Instagram, Foursquare and Flickr to the collector in order to enhance accuracy and size of the data.

Secondly the web application is very flexible and can be extended to allow the upload of .csv files which also would be added to the database and MapServer. This would allow TPS clutters to be exported and viewed in a high speed environment. There are many exiting things that can be done with the technology used.
In addition to these substantial ideas for future development there are several interesting new technologies, relevant for the project result that can be investigated. Because of the plug and play property of the components used in the project it is very easy to replace technologies. According to the author, interesting technologies and algorithms include MapD (Section 8.1), MongoDB (Section 8.2), ClusterHulls (Section 8.3) and Storm (Section 8.4).

8.1. MapD

This is an extremely interesting project both from a ingenuity point of view as well as from development and functionality. It is a database system, that still is under development and the plans are to release it as an open-source database. The special thing about this Database is its query speed. The creator claims that the first alpha version of MapD has a speed reaching up to 700,000 (seven hundred thousand) times that of a normal PostgreSQL.

The MapD project is displayed on multiple locations, most notably is the Twitter map that displays 300 million tweets and lets users query and filter the data in real time.

The secret behind the systems incredible speed is the usage of GPU:s in conjunction with the main CPU in order to calculate data. This is in comparison to conventional databases where all the processing is done by the CPU. The process relies heavily on parallelizing calculations between all available GPU-processors as well as the main CPU. Furthermore, because GPUs are built for visualizing and rendering data it gives the by-product of being incredibly efficient at displaying large data. On top of this GPUs generally have their own memory that is quite large (12GB per card * 8 giving total GPU memory of 96GB on the TwitterServer). This allows the entire database to be loaded on the GPU memory.

The cost is also something to consider, the server the developer of MapD is running the Twitter sample on has a total cost of about $5000. This is a comparably small price compared to other high performance servers.

More information about MapD can be found on the official homepage [11].

8.2. MongoDB

MongoDB is a NoSQL storage engine which has some spatial awareness with GEOMongo but it is still rather young. Therefore it is not supported by many other GeoSpatial applications and the implementation is not on the level of that of PostGIS. However even at this stage it could be a good option for storing raw data in MongoDB while keeping preprocessed data in PostgreSQL. This because NoSQL storage engines are better at dealing with big data than conventional SQL databases.

More information about MongoDB can be found on the official homepage [9].

8.3. ClusterHulls

This is the result of a research for handling the problems of streaming GeoSpatial data. The problems of streaming data is the "endlessness and speed of the incoming data which causes problems with both storage and processing of the data. ClusterHulls is an idea trying to overcome this by preprocessing and summarizing data in a way that limits the memory used in order to save and calculate unlimited data-streams.
The main idea revolves around creating convex hulls representing the spatial information contained in the data. These hulls are limited in number and size, the project discussed in this report could use them to find hotspots. This may be an interesting field to investigate but will most likely be hard to adapt so that it can reliably find hotspots in the entire world by using only a single set of parameters. It would most likely have to be used in smaller areas to make the configuration possible.

More information can be found on the ClusterHulls paper [12].

8.4. Storm

This is a tool that works with streaming incoming and outgoing data, it connects the incoming and outgoing sides through network protocol. This allows new data sources and new receivers to be added on the fly without disturbing any already running sources or receivers.

More information about Storm can be found on the official homepage [13].

References

[8] Twitter API Documentation: https://dev.twitter.com/docs
[10] Maximum access per twitter stream: https://dev.twitter.com/discussions/2655
[14] The web application is right now launched on the Ericsson Server http://10.44.91.87/twapp/
[15] All source code of the application is found in the repository /TwitterGeoLocation

Acknowledgments

First and foremost I want to thank Juan Manuel Melero, Javier Romero and Salvador Pedraza for giving me the opportunity to work for them at Ericsson with this project. It has been a pleasure living in Malaga and working on such an interesting and challenging project.
Furthermore I want to thank all the people who have helped me with small or large obstacles during the development process.

In particular I want to thank Jose Molina for his continued mentoring and guidance in Java and Database management. He has been an invaluable technical input when encountering large obstacles in the project.

I also want to thank Rosa Maria Javes who will take over the project after my return to studies, I am certain she will take good care of it.

Lastly I want to thank Åsa Schneider for her help rereading and commenting this report countless times.
Apache
The world’s most popular web server.

API
Application Programming Interface.

Cache
A temporary storage of data.

CPU
Calculations processing unit.

CSS
Cascading Style Sheets, styling language for HTML.

EPSG
European Petroleum Survey Group, they maintain a list of all geographic projections.

FastCGI
Fast Common Gateway Interface, protocol through which web servers can run programs, variation of regular CGI.

GeoSpatial
Information about the relative position of things on the earth’s surface.

GIS
Geographic Information System.

GPU
Graphics processing unit.

GRD
grid files, rasters that are used in GIS applications.

HTML
Hyper Text Markup Language, main component in web page development.

JSON
JavaScript Object Notation.

NoSQL
Not Only SQL.
**Overhead**

Refers to the resources required (often time) to set up an operation, these are often deemed as unrelated but necessary sub operations.

**SQL**

Structured Query Language.

**TPS**

Trace Processing System, Ericsson network optimization tool.

**Web Framework**


**WKT**

Well Known Text.

**WMS**

Web Map Service, a service serving any kind of map images to a web client.

**WMTS**

Web Map Tile Service, a service serving map tiles to a web client, usually in .png.
Populärvetenskaplig Sammanfattning

I dagens samhälle är sociala medier en av de viktigaste informationskällorna. Detta gäller naturligtvis för privatpersoner men framförallt kan företag dra stor nytta av det stora informationsflödet. Företag använder den uttömliga strömmen av data till all möjlig statistisk analys som underlag och stöd för att ta såväl strategiska som operativa beslut. I detta arbete används informationen som finns från Twitter rörande användarnas geografiska position. Syftet är att därigenom underlätta Ericssons beslut rörande optimering av mobila nätverk.

Trots att informationen den insamlad datan innehåller enkelt och snabbt kan förmedlas på kartor ställer datamängdernas storlek till en del problem. Detta leder till ett behov av karttjänster med interaktiva kartor innehållande mycket data som trots det kan zoomas och förflyttas i en hög hastighet. Hastigheten är viktig i sammanhanget då man inte vill sakna användarens arbetsflöde vid användande av tjänsten.

Projektet innefattar en fullständig produkt, från insamling av data från Twitter, lagring och förbehandling av dessa till och med den slutgiltiga karttjänsten vilken är en interaktiv karta som kan zoomas och förflyttas i hög hastighet. I skrivande stund visar kartan 700 millioner datadioducer över hela världen och bearbetar zoomning och förflyttning inom loppet av några sekunder.

En av dem stora fördelarna med designen och valet av teknologier i projektet är att det enkelt att lägga till nya datakällor till kartan. Det i sin tur öppnar upp många nya möjligheter för Ericsson att bygga vidare på projektets resultat. Optimeringsingenjörer kan exempelvis ges större överblick över datamängdernas information innan de importerar och utvärderar dessa i de förhållandevis tröga optimeringsprogramvarorna.