Spatio-temporal Analysis of Twitter Data Streams

Michael Pryor
1002157
Supervisor: Mirco Musolesi

Submitted in conformity with the requirements
for the degree of BSc Computer Science with Industrial Year
School of Computer Science
University of Birmingham

Contents

Chapter 1

Introduction

Social networks have rapidly grown in importance in recent years with the two of the most popular social networking websites, Facebook and Twitter[1], seeing large rises in usage year on year. For example, Facebook's number of active users rose from 100 million in 2008 [Zuckerberg, 2008] to over 1 billion in 2012 [Fowler, 2012]. Twitter generates 340 million tweets daily (1 billion every 3 days) and has achieved this in only 6 years of operation [Twitter, 2012]. Social networking websites operate across geographic boundaries with information flowing around the globe with virtually no restriction. Relationships such as Facebook friends and Twitter followers create networks of linked users in which information flows; these networks vary in size from hundreds to millions and have interesting properties which we can analyse [Rainie et al., 2012].

Most social networks have a geographical aspect in the form of a real world coordinate or user provided location field [Takhteyev et al., 2012]. As discussed by Lima and Musolesi, we can use this to analyse information dissemination from one location to another, and to identify important users who have significant influence on target locations [Lima and Musolesi, 2012]. There are a multitude of real world applications for this analysis, including:

- Targeted information spreading: We can target geographic areas to disseminate information in by prioritising users by the number of contacts they have in the target area; these users are likely to be more influential, and therefore more useful for our purposes [Lima and Musolesi, 2012]. By targeting the most influential users it is hoped that information disseminates to the target audience through such mechanisms as Twitter retweets or Facebook status sharing [Li and Shiu, 2012]. This can be applied to help instigate or control regional events such as riots, natural disasters or targeted advertising campaigns [Lima and Musolesi, 2012].

- Economic analysis: We can predict changes in demand for goods and services by analysing links between cities and countries. If an item becomes popular in London for example, where is demand most likely to spread? As detailed by Scellato et al. a practical application for this is online content delivery (e.g. YouTube). In such systems, content is duplicated across multiple servers geographically with the aim of minimizing distance between the content provider and consumer. Performance can be improved by caching locally prior to a predicted spike in demand for a specific set of content [Scellato et al., 2011].

One key requirement for online social network analysis is data; without data no analysis can be done and no conclusions can be drawn. It was decided that the goal of this project would be to produce a scalable system for retrieving large quantities of online social network data including information about users and their activities. The system should provide visualisations of both historical and real time data, as well as provide influence analysis as outlined in the paper by Lima and Musolesi.

[1]Most popular is defined by estimated unique monthly visitors. Based on ranking as of 12th April 2013: `http://www.ebizmba.com/articles/social-networking-websites`

Chapter 2

Background

Social networks and social media in general have grown to become a critical means of communication throughout the world and are becoming increasingly important as a tool for organizations to communicate with existing customers and advertise to prospective customers. Kaplan and Haenlein detail the importance of various different types of social media including blogs, collaborative projects, content communities and social networking sites [Kaplan and Haenlein, 2010]. Heidemann et al. explain many potential uses for social network analysis from a business perspective including applications in fields such as marketing and sales, customer services and human resources [Heidemann et al., 2012]. The rise of social networks has lead to the creation of social media analysis companies e.g. Beevolve is a company which specialises in analysing social media for small businesses [Beevolve, 2012]. Algorithms have been developed in order to analyse users and their prominence within social networks e.g. Li and Shiu looked at information dissemination specifically for the purposes of improving targeted advertising campaigns [Li and Shiu, 2012].

Research has been done into the use of geographic information associated with user accounts and user generated content to improve quality of service in areas such as targeted content delivery and information dissemination. Scellato et al. analysed social network usage to improve content delivery network cache replacement policies, caching content closer to consumers [Scellato et al., 2011]. Lima and Musolesi propose several methods of calculating the influence of a user on a target location [Lima and Musolesi, 2012].

In order to utilise geographic information in social networks, there must be some way of identifying a user's real world location. This usually involves processing user provided fields or by parsing geocoding done directly by the social network. For example, Takhteyev et al. looked at the impact of factors such as geographic distance, travel and language on social neighbourhoods of Twitter users; they manually geocoded a selection of users by processing the user provided location field associated with Twitter user accounts [Takhteyev et al., 2012]. Several research articles have used online geocoding to process geographical data and have done so with reasonable accuracy. Cao et al. used the Google geocoding API to convert geographical coordinates into real world locations [Cao et al., 2010]. Lima and Musolesi used the Google geocoding API to convert Twitter location fields into real world coordinates [Lima and Musolesi, 2012]. It is also possible to geocode users simply by looking at the contents of their messages; Hecht et al. used a machine learning technique to geocode Twitter users by analysing the contents of their tweets [Hecht et al., 2011].

In the process of analysing social networks, researchers have independently built systems for retrieving, geocoding and analysing data from social networks. These systems often share features and functionality which are duplicated between systems. The aim of this project is to reduce such duplication of work and provide a central source of real time and historical social networking data retrieval and analysis.

Chapter 3

Problem Analysis and Solution Design

3.1 Online Social Networks

3.1.1 Choice

A number of online social networks provide tools for receiving information about users and their activities. It was decided that Twitter was the most functional choice for the purposes of this project for a number of reasons.

The Twitter streaming API allows us to subscribe to a geographical area or a set of keywords and retrieve tweets and information about the users which created the tweets in real time with high throughput [Twitter, 2013e]. From our tests, Twitter provides a base level of access which provides approximately 180,000 tweets per hour, per stream (see Figure A.1 on page 26). It achieves this using a long polling HTTP request which keeps the connection alive and streams data as it arrives [Twitter, 2013b]. In comparison, Google+ and Facebook require users to manually poll by repeatedly establishing an HTTP connection and requesting data in groups of 20 and 25 respectively [Facebook, 2013b; Google, 2013b]. This means that data cannot truly be processed in real time as there is some delay between each poll. Furthermore, the overhead of re-establishing the connection and re-querying combined with the small amount of data retrieved with each request means that these APIs may not scale up as well as Twitter's streaming API in terms of performance and throughput. Moreover, it is likely that some form of rate limitation may be reached (although these limits are not well documented), especially when querying multiple different filters simultaneously.

A second major advantage of Twitter is that its users tend to post far more public data than other social networks such as Facebook. While Twitter does provide mechanisms to protect Tweets, it is by design a more open platform encouraging users to share with the world rather than with a specific network. Facebook and Google+ place an emphasis on privacy which leads to the creation of networks of friends which interact only with each other. Posts which are protected in this way are not accessible through APIs without express user permission. Beevolve, a commercial social media analysis website, did a study of 36 million Twitter users and found that approximately 11.84% of users had protected accounts while the remaining 88.16% were fully accessible [Beevolve, 2012]. In contrast, research into Facebook privacy settings has found that the majority of users hide significant amounts of information from non friends. Yamada et al. surveyed 149,862 profiles and found that only 12.6% had their basic profile information publicly available (such as gender and age), and only 7.8% had their news feed (posts and statuses) publicly available [Yamada et al., 2012]. Stutzman et al. found similar results analysing 1066 profiles over several years to find only 21.5% of users sharing interests publicly and 13.2% sharing their birthday publicly in 2011 [Stutzman et al., 2013]. This limitation of publicly available information means that to properly mine data with Facebook (and other similar online social networks) one should request authorisation from users to access their otherwise protected information. The Facebook and Google+ platforms readily support this but rely on the direct participation of users [Facebook, 2013a; Google, 2013c].

In summary, Twitter's streaming API provides a readily available high throughput feed and does not suffer from high levels of restrictive user privacy settings. In comparison, the user base of Facebook and Google+ is proportionally less open to applications and requires additional work in requesting user permissions if this issue is to be circumvented.

3.1.2 API

Optimally using the Twitter streaming API raises a number of issues. Firstly, with over 180,000 tweets per hour, per stream (see Figure A.1 on page 26), we must make sure to build a scalable system which can deal with fluctuations in load and for which we can easily add capacity should the rate limit change. In the event that we receive more data than we can handle, we must gracefully deal with this by dropping tweets or safely persisting them. It is worth noting that Twitter forcefully disconnects subscribers which do not consume the Twitter feed fast enough and fall behind [Twitter, 2013b].

A second consideration is that any analysis we perform should take into account that we are rate limited, sometimes to the extent that we are only receiving a small fraction of the total tweets in the stream. Figure A.2 and A.3 (on page 27 and 28) show this graphically. Table B.1 shows that only 10.57% of tweets in the stream were received over the entire 70 hour period. While this may seem poor, it should be noted that we received over 12 million tweets in that period; many lower volume streams can be completely consumed without rate limitation. Moreover, to consume the additional tweets would require substantial hardware which is not available in the development of this project.

For any geographical analysis of Twitter data, we will need to associate a geographical location with each user and/or tweet. Twitter allows for geotagged tweets by using information from GPS enabled devices such as smart phones [Twitter, 2013d]. These devices can send tweets which are associated with a specific geographical coordinate; we can use this information to pin point their real world location. In addition to this, Twitter attempts to convert this coordinate into a named location with some additional information such as what type of place it is e.g. country or city [Twitter, 2013c,d]. This process of converting a coordinate into a named location is known as reverse geocoding [Cao et al., 2010]. This field may also refer to a real world location which the user is talking about in their tweet; we can use this information to infer a relationship between the location and the user [Twitter, 2013f].

For coordinate and geocode information to be available in this way, the user must be using a device which is capable of providing the information and in addition to this the user must have enabled this feature on their account [Twitter, 2013c]. As a result proportionally few tweets have this information embedded in them. Figure A.33 (on page 48) shows how on average only 1.33% of tweets received from the Twitter streaming API on a high traffic stream over a 70 hour period were geotagged by Twitter. However, figure A.33 also indicates that we can greatly improve the proportion of the stream which we are able to consume by processing user location fields which 59.3% of user accounts had associated with their account.

Many user accounts have a user provided location field which is intended to indicate their real world location [Twitter, 2013f]. As it is user provided, its nature can be unpredictable. Takhteyev et al. found 84.9% of a sample of 3360 users had useful information in their location field, with only 15.1% with a "very broad, non-spatial, humorous, or undecipherable" location field [Takhteyev et al., 2012]. To use this effectively we will need to convert the location field into a more meaningful form, associating it with a specific city, town, country and/or continent and group users together in this way. In the next section we explore how best to do this.

3.2 Geocoding

3.2.1 Method of geocoding

Geocoding is the process of converting a named location into a geographic location e.g. coordinate [Roongpiboonsopit and Karimi, 2010]. Geocoding is a non trivial task with a number of complex problems discussed in this section. What is the best way to geocode effectively? One option is to build a geographic information system which can convert Twitter user location fields into real world locations; there are a wide range of freely available datasets available to help us do this, e.g. TIGER [Drummond, 1995; United States Census Bureau, 2013]. While this approach may be feasible, it is not necessary as there are a large number of free to use online geocoding APIs with simple interfaces. Using an online geocoder is good from a re-usability stand point and greatly reduces our development time line [Roongpiboonsopit and Karimi, 2010].

A number of geocoding APIs exist which provide HTTP based services for converting locations into coordinates. When choosing which geocoder to use accuracy was our main consideration because the quality of the geocoding results will have a direct and significant impact on the quality of any geographical based analysis. Geocoding APIs vary in accuracy significantly and some have problems with non geographic information. When geocoding Twitter location fields, many users may have arbitrary nonsense stored as their location. It is therefore important that the geocoder handles this and does not produce false positives. Hecht et al. highlights the problems that false positives can cause, finding that of 1380 non geographic location fields 82.1% were geocoded successfully using Yahoo geocoder; this should have been 0 [Hecht et al., 2011]. Roongpiboonsopit and Karimi compared 5 of the major geocoders including Geocoder.us, Google, MapPoint, MapQuest and Yahoo. The report suggests that Google, Yahoo and MapPoint are of similar quality while MapQuest is less accurate and Geocoder.us is less accurate still [Roongpiboonsopit and Karimi, 2010]. Much previous related research has used the Google geocoding API to successfully process Twitter location fields or reverse geocode coordinates and as a result the Google geocoder was chosen as the primary source of geocode information [Lima and Musolesi, 2012; Cao et al., 2010]. Work was also done using OpenMapQuest because it is powered by open source data [OpenMapQuest, 2013]. In order to accommodate both geocoders a configuration switch was introduced in the configuration file to switch between each; only one geocoder is used at a time.

There are a number of issues that were encountered while using both the Google and OpenMapQuest geocoding APIs. Table B.2 (on page 49) shows a number of possible location fields which should all geocode to the same location. The first issue is how we differentiate between locations which have the same name. For example, if the location field is "London" does the user reside in "London, United Kingdom", "London, Kentucky, United States", "London, Ohio, United States" or "London, Ontario, Canada". There is no perfect solution to this problem since the location field is ambiguous, but we can use location biasing to improve accuracy. Location biasing is some method by which locations are ordered in preference. Previous research has shown that followers are more likely to be close to the followee geographically [Takhteyev et al., 2012; Lima and Musolesi, 2012]; as a result we can order possible geocode results by distance from the followee, choosing the nearest.

A second problem that should be considered is how to deal with web text i.e. text which is not grammatically or textually correct from a language perspective. For example, "L-O-N-D-O-N". A similar issue is that of extracting the location from a string of text such as "I live in London with my friend"; how should we extract the location meaning from this sentence? Further complex sentences with negation may arise, for example "not Cardiff, now live in London"; here we should be careful not to wrongly geocode the user as located in Cardiff when in fact the sentence indicates that they now live in London. Table B.5 shows how this solution was designed. We ignore punctuation and case to form the word "magic" from "M-A-G-I-C" and we reduce white space to single spaces to avoid differentiating by features which are unlikely

to impact meaning. We then call all combinations of the words in the input until we find a match. This helps to extract potentially meaningful information such as "magic land" in the first case or "London" in the second case. To avoid expensive searches we limit the number of words per query, in this case to 4. This approach favours results with more words, for example "London, UK" is searched for before "London" or "UK" so that we maximize precision. This is particularly important with cases such as "New York City"; an alternative solution would be to query each word individually, we would start with "New" and fail, and then query "York" which may resolve to "York, UK" which would be wildly inaccurate as the intended target is probably "New York City, US". While our solution does not handle the negation case alluded to earlier, it is an improvement on geocoding the location field as is, as we are able to extract location data from sentences, and are able to handle cases where grammar is used improperly e.g. "L-O-N-D-O-N".

3.2.2 Geocoding with high throughput

Google rate limits its geocoding API usage to 2,500 requests per day, which works out at 104 an hour or roughly one every 35 seconds [Google, 2013a]. OpenMapQuest is not strictly rate limited, but through consultation with them they recommended not querying more than once a second to avoid over saturating their servers. We need to process at least 50 tweets per second with a throughput of over 180,000 tweets per hour (see Figure A.1 on page 26) so these rate limits are insufficient to directly use their geocoding services to process every tweet.

Therefore, we must cache the results of our geocode queries in some way so that repeat queries can be done without using an external geocoder (such as Google or OpenMapQuest). To accommodate this we have two levels of caching. First, we cache the results in a database mapping the query to a result from the geocoder. Second, we have a least recently used in memory cache of the same form. The cache holds the most recent queries so that frequent queries can be done cheaply and with low latency by reducing the number of calls made to the database. By reducing load on the database we can improve throughput as the database is used heavily elsewhere in the project. This in memory cache has a configurable maximum size to avoid exhausting memory of the host. Least recently used was chosen as it has been thoroughly evaluated by previous research to find that it is an effective way of caching, for example Jelenkovi and Radovanovi investigated the efficiency of least recently used caching on web crawling [Jelenkovi and Radovanovi, 2004].

Figure A.4 is a flow chart showing how a tweet is geocoded, with three distinct stages: from memory, from database and from online geocoder. Each one runs in parallel; for example, while a tweet is being geocoded by the online geocoder (which can take several seconds), other tweets can be geocoded. The aim is to maximize the number of tweets geocoded, and to limit the need to drop tweets as a result of being unable to geocode the tweets fast enough. Table B.3 shows that over a 70 hour period 99.84% of tweets were geocoded using the in memory cache while only a small percentage of tweets were successfully geocoded from database or from the external online geocoder. This is not surprising since we throttle the database and online geocoding, but it does give some idea how much throughput the in memory cache is handling.

3.2.3 Cities, Countries and Continents

The focus of the geocoding solution so far has been on geocoding cities and large towns. However, we also want to look at influence and data at a country and continent level. Online geocoders give us results that contain brief information about the city and where it is located, but do not give much (if any) detail on the country which the city belongs to. Specifically, we need to know the geographic coordinate, bounding box and full human readable description of the country and continent that the city belongs to. This additional information helps us to display results to users, for example on a map. In the case of the Google geocoder we receive the country

name e.g. “United Kingdom” and an ISO 3166 country code e.g. “GB” with cities which we geocode [Google, 2013a]. We are missing coordinate, bounding box and continent information.

There were a number of options considered for solving this problem. First we looked at using bounding boxes of countries. We already know the coordinate of the city as this is included with geocode results, and if we know the bounding boxes of countries then we can simply find the box which contains the coordinate associated with the city. Such bounding boxes and other data that we need are freely available from sources such as the geographic names database [United States Board on Geographic Names, 2013]. Unfortunately, this does not work well in practice as there are a number of countries which overlap significantly, for example figure A.8 (on page 33) shows how nearly all of Bosnia's bounding box is within Croatia's, making it difficult to associate the correct country with cities of Bosnia. Further problems exist when a country's territory includes islands or other land a large distance from its mainland. For example, Spain includes the Canary Islands and as a result its bounding box overlaps with Portugal and Morocco (see figure A.9 on page 33).

An alternative solution which we decided to implement is to use the ISO 3166 country code associated with records in the geographic names database. We link this with the country code provided with online geocode results. This is sufficient for our needs because both Google and OpenMapQuest provide this country code with the results of geocoding cities and large towns [Google, 2013a; OpenMapQuest, 2013]. Moreover Michael Collinson from OpenMapQuest has already been through the process of sanitizing data from the geographic names database to produce a CSV file containing exactly what we need for the purposes of this project [Collinson, 2009]. We load this CSV file into memory and can match cities by ISO 3166 country code to countries with virtually no overhead as it is simply an in memory lookup. To tackle the issue of continents, the CSV file lists the continent that each country is associated with but without details of the continents' bounding box or coordinate. As there are a very small number of continents we simply worked out by hand the bounding box and coordinate of each continent and put them into a CSV file. The project then loads this CSV file into memory and matches the country to its corresponding continent.

This solution works well but suffers a number of drawbacks. Turkey and Russia span across two continents; Asia and Europe. We have no way of correctly handling this situation since the country code is static regardless of which part of the country a city is located. The work around for this is to choose one of the continents to use, but this is not ideal. A further drawback is that this method requires housekeeping as country boundaries and names occasionally change. These changes will not be reflected until the CSV file is updated. On the flip side, we have complete control over the CSV file and can make corrections as necessary.

3.3 Influence

3.3.1 Calculation

Lima and Musolesi detail how ranking users by number of ties they have within a network is a simple indicator of influence known as “spatial degree centrality” [Lima and Musolesi, 2012]. In Twitter a tie can be defined as a user following another user, and for our purposes, a network can be defined as a selection of followers who are associated with a target location.

At a user level, Lima and Musolesi detail a formulae for defining the influence of a user on a target location. This formulae is [Lima and Musolesi, 2012]:

$$C_{i,S} = \sum_{j \in N_i} \mid P_j \cap S \mid \tag{3.1}$$

$C_{i,S}$ is the influence of user i on a set of target locations S. N_i are the followers, or neighbourhood, of user i. P_j is a set of locations which follower j is associated with. Therefore, if

the cardinality of S is 1 (there is only 1 target location) then $C_{i,S}$ amounts to the number of followers user i has who are associated with the location S; Lima and Musolesi describe this as the "size of the audience of user i in the region" [Lima and Musolesi, 2012].

This measurement does not normalize for variation in size of neighbourhood (number of followers) and does not take into account other potentially relevant information such as retweets. Despite these limitations, which are evaluated in section 5.3, spatial degree centrality still provides a useful measure of influence [Lima and Musolesi, 2012].

For the purposes of this project it was decided that we would focus on spatial degree centralities with just one target location. We could look for example at the influence a user has on London and Tokyo, but do so separately to produce two separate degree centralities. This simplified the solution design but left room for future development. The second goal of this project beyond looking at individual users is to look at the influence of cities, countries and continents on each other. This is achieved by totalling the spatial degree centralities of users associated with a source location to get for example, the spatial degree centrality of London on New York City.

A final topic of interest is to look at how influence changes over time. Previous research suggests that neighbourhood structure does not change very frequently on the whole [Lima and Musolesi, 2012], but the type of people talking about a topic can do, and this would allow us to analyse how the neighbourhood structure of users talking about a topic changes over time. This is particularly relevant when dealing with influxes of people for a localized event. For example, a football stadium may draw fans to a stadium in Manchester, it could be interesting to analyse how the neighbourhood changes in response to the influx of fans, and how the audience of users in Manchester tweeting about football changes.

3.3.2 Architecture

The Twitter streaming API does not provide follower enriched user data with tweets. For the purposes of this report, a follower enriched user is defined as a user with follower IDs and follower data loaded. An unenriched user has only primitive details like number of followers, but does not have specific per follower data. As a result of this limitation of the streaming API, we must make a separate API call to enrich follower information. Twitter provides two HTTP end points; the first allows us to retrieve follower IDs of a user as a list e.g. 1500, 1501, 1600; the second allows us to convert a list of user IDs into a list of user data objects [Twitter, 2013g,h].

At the time of writing, both are rate limited to retrieving 5000 follower IDs of a user once a minute and retrieving user data from 100 IDs once every 20 seconds [Twitter, 2013g,h]. So for example, it will take at least 1 minute to retrieve follower data of a user with 300 followers. Four API calls need to be made, one for retrieving 300 follower IDs, and three to retrieve user data for these IDs in batches of 100. As a result we cannot usually enrich follower information for all users that create tweets which appear in our stream; except for very low traffic streams. To solve this problem we need some way of selecting users to enrich. It was decided that two layers should be introduced. First, users should be able to choose the source location that they are interested in; referring to formula 3.1 (on page 8), this would involve selecting one or more acceptable values for S. This reduces the search space and enables the user to extract the data which fits their requirements. Secondly, we should limit the number of users waiting to be enriched, to prevent exhausting memory if a source location is high traffic. Both layers should pass through tweets of users which we do not enrich, as they are still used down stream by other components of the system.

A second problem is that of handling users with varying numbers of followers; users can potentially have several million followers and we need to ensure that our system can handle this and will not run into memory exhaustion or performance problems. Figure A.10 (on page 34) is a flow chart showing how the follower enrichment module is designed to combat this. The crucial element is that when processing user data items, we geocode and output the follower

before moving onto the next batch. This allows us to store only the current batch of 100 followers in memory at any one time. Moreover, the data flow out of the module is kept consistent which prevents down stream modules from being flooded with data. For example, if we processed all 1 million followers of a user before sending any results down stream, down stream modules would receive a large burst of data at the end of the follower enrichment process, which would likely cause performance issues while this is processed.

3.3.3 Temporal

In order to analyse how neighbourhoods of a Twitter stream change over time, there must be some method for efficiently storing and accessing data for arbitrary periods of time. We aimed to design and build a flexible solution which allows users to select any start and end date/time, and view the spatial degree centrality of neighbourhoods within that time period. In order to store large time ranges with reasonable granularity it was decided that we should store our data in a database; we assume that our database is properly indexed such that read/write performance is logarithmic in complexity $\mathrm{O}\left(\log_2(m)\right)$ where m is the number of records in the index.

A naive implementation would be to break the data set into a series of time steps of for example, 1 minute each, and store centrality data retrieved since the last time step in a new database record. Over time more time steps would be created and we would have one time step record per minute since the Twitter stream was first created. To determine the centrality of a time period, we would simply have to combine all time steps within the specified time range to form a single centrality result. This process would involve iteratively looking up time steps and thus has linear complexity; i.e. if there are 16000 time steps in the time range then we must do 16000 database read operations. When taking into account the expected time complexity of database read operations mentioned earlier we see that searches have a linearithmic complexity of $\mathrm{O}\left(n \cdot \log_2 m\right)$ as shown in table B.6 (on page 52). This does not scale well with big data sets which may have several million items.

To solve this it was decided that binary trees should be used. Table B.6 shows the performance improvement gained by using a binary tree structure; we go from linearithmic to logarithmic read complexity but write performance is slightly reduced as we now have to maintain a tree structure with every write operation. Figure A.11 (on page 35) shows an example of what the tree structure may look like. Each location of each Twitter stream has a separate tree which describes the influence of that location on other locations. The binary tree segments time where broadly the left half of the tree describes the first half of the Twitter stream's life time and the right half of the tree describes the second half of Twitter stream's life time. Table B.6 shows an example where we have 16384 time steps and shows a best case performance of 15 operations to read a record (1 database read); this would occur when reading the root node, for example in figure A.11 reading the time range 0 to 8. Table B.6 shows a worst case read performance of 450 operations where a node at each level of the tree must be read; figure A.13 shows an example of this visually.

Our solution has a number of interesting extensions over a standard binary tree; these extensions allow us to more efficiently manipulate the tree in a database by minimizing database write and read operations. There are two main optimizations, one for reading and one for writing.

In order to improve update time we do not maintain just 1 single tree per location, per stream. Instead we maintain multiple separate trees which are combined together at a later time step. Of these separate trees, only the newest rooted tree is updated with new influence data. This tree has one node and so updating its value requires only one database operation because no parent nodes need be updated (since none exists). When a new time step is created as a rooted tree, previous time steps are combined together into single trees where possible. Previous time steps are historical and thus will not be written to so the height of historical trees will not impact performance of future writes. It should be noted however that performance when combining trees is logarithmic, so worst case performance is not improved. This process

is shown in figure A.12 (on page 36) where until time step 4 there is one tree, but at time step 4 a second tree is created, and then at time step 6 and 7 a third and fourth rooted tree are created. Then at time step 8 all four trees are combined together to form a single tree containing the same data as the original trees. When updating the value in place without merging trees, best case complexity is achieved with only one node being updated. With the use of database indexing this can be done with a direct lookup of complexity $\mathrm{O}(\log_2 m)$. When merging trees worst case complexity is achieved if a new root node is required and all trees are reduced to one single tree, as shown in figure A.12 when moving to time step 8. In this case we need to create a new parent node at every level of the tree. The height of the tree is $\mathrm{O}(\log_2 m)$ and each new node requires $\mathrm{O}(\log_2 m)$ time to insert, hence the worst case complexity is $\mathrm{O}(\log_2 m \cdot \log_2 m)$. The importance of this optimization depends on the frequency of updates and on the length of time steps. If a time step is 1 hour for example and we are frequently generating centrality data then there will be a large number of in place changes resulting in best case complexity. However if the time step is 1 second then we will be frequently merging trees and the potential performance improvements will not be realised.

Currently our searches are precise; for example, if our time steps are 1 minute in length but we have a year's worth of data then our search will not complete until it has retrieved a result which is accurate with minute precision. For most purposes however, if searching for such large time range, it is unlikely that individual minute precision is necessary. We can greatly reduce complexity by introducing a maximum number of individual lookups per search. In this way we read large time ranges first and work our way into smaller units until we reach the cap. Our results will be less accurate but computation time will be significantly reduced. This method ensures that time ranges are treated similarly, for example if searching for an hour time range then loss of accuracy may result in seconds being lost; if searching for a year time period then loss of accuracy may result in days being lost. In figure A.13, if we wish to retrieve data for the time range 1 to 31 we would have to do 8 lookups (as marked in the diagram) to be 100% accurate. To reduce computation time we could cap the number of lookups to 4 and retrieve the 4 coloured nodes closest to the root. This would yield results which cover 75% of the search space and halve computation time. This optimization reduces the worst case complexity of the binary tree to $\mathrm{O}(x \cdot \log_2 m)$ where x is the maximum depth and m is the number of nodes in the tree (see table B.6); the best case remains unchanged because it covers the case where the root node covers the entire search space and so is the only node that is accessed.

A final consideration is the amount of storage that the tree consumes. We are currently dealing with full and complete binary trees with a defined shape and nodes which have exactly 0 or 2 children [Shaffer, 2001]. Full and complete binary trees possess properties which allow us to analyse storage requirements for the general case [Shaffer, 2001]. Shaffer proves that "The number of leaves in a non-empty full binary tree is one more than the number of internal nodes" [Shaffer, 2001] where an internal node is a node with 1 or more children, and a leaf node is a node with 0 children [Shaffer, 2001]. This amounts to an equation $L = I + 1$ where L is the number of leaf nodes and I is the number of internal nodes. This can be rearranged to $I = L - 1$ and thus $I + L = 2L - 1$ meaning the total number of nodes is $2L - 1$. Every leaf node corresponds directly to a time step and no time step is without a leaf node. As a result, this formula can be used to predict the total number of nodes in the structure, given the number of time steps that have passed. This may not represent the actual number of nodes in the data structure because of optimizations discussed earlier, but it will represent the maximum possible number of nodes. The storage requirements seem reasonable and can be expected to rise linearly over time.

We can however optimize this storage usage. If a low traffic stream is being consumed and we rarely generate spatial degree data then in the current design we will write a large number of empty leaf nodes to the database. Clearly these nodes are redundant and we can therefore reduce storage usage by leaving them out. However, we must be careful to properly merge

trees together as appropriate to avoid inadvertently hiding useful nodes from their prospective parents; if not merged properly we may end up with a number of rooted nodes left dangling and unreadable by most searches. Figure A.14 (on page 37) demonstrates the reduction in number of nodes as a result of this optimization. Figure A.14 also highlights several parent nodes which must be constructed to avoid dangling data. These parent nodes would normally be built in the tree merging process at the same time as the omitted leaf node. Since the leaf node is omitted the tree merging process will not commence; to counter this we wait for the next leaf node to be created and then ensure all past parent nodes which could contain data are built, even those with just one branch.

3.4 Front End

3.4.1 Web based

For the display of information and for interaction with users, some form of graphical user interface is required. Increasingly, rich applications are being developed with the use of HTML and JavaScript web based front ends; these applications rely on new technologies such as HTML5 to provide functionality which would normally only be possible from a desktop application [Mikkonen and Taivalsaari, 2008; Lawson and Sharp, 2011]. Web based applications have a number of advantages over traditional desktop applications.

First, barriers to usage are lower; users can begin using the application straight away with no installation or other barrier to entry [Farrell and Nezlek, 2007]. Second, websites can be more easily shared via social media and search engines [Kaplan and Haenlein, 2010; Shih et al., 2012]. Users can send specific pages to their friends or post links on social networking websites [Kaplan and Haenlein, 2010]. Search engines crawl through the web and direct users to content via search results, increasing the likelihood of our application being noticed and helping us to build up a user base [Shih et al., 2012].

A second advantage of web based application development is platform independence. Websites run in an environment which is controlled by the web browser; most web browsers will display pages independently of platform e.g. on Linux a website should (in most cases) work in exactly the same, or a very similar same way as on Windows and visa versa [Mikkonen and Taivalsaari, 2008]. By relying on the browser to fulfil our portability requirements, we are reusing functionality and reducing our development and testing time line. Note that web design does suffer from difficulties in maintaining consistency between different web browsers, but with the advent of HTML5, standards are being more clearly defined and such inconsistencies becoming less common [Lawson and Sharp, 2011; Anttonen et al., 2011]. In addition to this, third parties have been developing substantial extensions which facilitate cross browser development e.g. jQuery extends JavaScript to provide cross browser functionality for a number of common operations such as HTML manipulation [w3schools, 2013a].

In summary, web based front ends provide a cross platform method for interacting with users. Users can begin using the system with little in the way of installation or setup procedures and the system can more easily benefit from social media and search engines to quickly build a user base. For these reasons it was decided that a web based front end should be developed.

3.4.2 Static vs Dynamic

Broadly we can separate web server usage into two generic types of request. First there are static requests which request information which does not change frequently (if at all), for example a normal web HTML page [Iyengar and Challenger, 1997]. Web servers usually cache these pages in order to improve performance as shown by figure A.15 (on page 38) [Iyengar and Challenger, 1997]. Research has shown that such caching schemes greatly improve performance by reducing computation time and I/O latency when reading from a storage device [Markatos, 1996]. Luckily

many web server technologies already implement some form of caching mechanism which is sophisticated enough for our needs, so we should not need to implement our own caching system e.g. Apache HTTP Server [Apache, 2013].

The second request type is for requesting dynamic data; dynamic data is data which we expect to change, often frequently [Iyengar and Challenger, 1997]. A number of key issues arise when handling dynamic data. The first issue is how do handle live updates. For our system, we may wish to show clients[1] tweets in real time. Using simple static HTML web pages alone the client would have to refresh their page to receive new tweets. Not only is this undesirable from a usability stand point, but this will increase load on our servers through repeated HTTP requests.

HTML5 web sockets allow a client to send and receive data in real time without reloading the page [Lawson and Sharp, 2011]. Web sockets are a bidirectional communication channel and can remain open indefinitely [Lawson and Sharp, 2011]. We can therefore use web sockets to stream tweets to clients as soon as they arrive in true real time. A naive implementation on the server side manages each connection individually with separate logical threads (see figure A.16 on page 38). This implementation works well but is not optimum in terms of performance because there is no caching. For example, if two clients request the same dynamic content then that content will need to be generated twice and may require expensive file system or database I/O to do so. Iyengar and Challenger show the performance benefits which can be achieved through caching of dynamic content [Iyengar and Challenger, 1997]. Standard web based implementations do not normally provide pre-built caching mechanisms for dynamic content because it is difficult or impossible to cache such data effectively without knowledge of how and when the data is likely to change [Iyengar and Challenger, 1997].

For the purposes of this project we designed a system for caching dynamic content which is based on the publisher/subscriber model. This model has been heavily researched and implemented in a variety of real time applications [Rajkumar et al., 1995; Kaiser and Mock, 1999]. In the publisher/subscriber model subscribers register with a publisher and receive all subsequent messages published by that publisher; subscribers do not interact with each other and publishers do not know about individual subscribers [Rajkumar et al., 1995].

In our solution, a subscriber is a web socket; i.e. a persistent bidirectional connection from a single client to the server [Lawson and Sharp, 2011]. A number of "web socket groups" act as publishers which web sockets subscribe to; a web socket group maintains a cached copy of some dynamic content and updates it when new real time data is received. Figure A.17 (on page 39) shows a diagram of the dynamic data caching system. Web socket groups publish real time updates in response to signals from the data provider. The data provider is an observable tree like structure which acts as a publisher; this is shown in figure A.18 (on page 40). Web socket groups subscribe to the parts of the tree which they are interested in, we describe these parts of the tree as "topics". As the tree updates it publishes the changes in data to subscribing web socket groups. The web socket group acts as a bridge from the data provider publisher to the web socket subscribers and converts raw data into information which is useful to the client.

When a client connects and requests dynamic content for the first time a web socket group is created and begins streaming real time information from the data provider to the client's web socket. When the web socket group is first created it may perform some setup operations, for example it may retrieve a set of cached tweets; these operations are often expensive and involve some form of disk or database I/O. When a second client connects to a topic which is already in use, it registers itself with the web socket group and receives a cached copy containing a snapshot of information based on recently received real time data on that topic. The snapshot sent upon the client registering with the web socket group ensures that the client's view is

[1]Throughout the report we often refer to users of the system explicitly as "clients"; we do this to avoid confusion between the notion of a Twitter user which we are processing, and the notion of a user of the system which we have built.

similar or identical to the other clients of the same web socket group. The snapshot is cached in memory and so it is relatively cheap in terms of CPU usage to send this to new clients. Connected clients receive identical real time updates from the web socket group and update their display accordingly. Finally, when web sockets disconnect they unsubscribe from all web socket groups. A web socket group with no subscribers is cleaned up; this is necessary to prevent memory leaking.

3.4.3 Tunnelling

A specific use case requires functionality which is similar to that of the static and dynamic request types discussed earlier, but contains certain key differences. Our project is a data collection tool and so users need to download data that the system has collected. We may have many gigabytes worth of data covering millions of tweets and users; how can our server best provide this?

We experimented with both static and dynamic requests to facilitate large file downloads. Static requests are not designed for file downloads because the entire contents of the file would have to be loaded into memory and then transferred. We cannot load many gigabytes worth of data into memory without risking memory exhaustion so it is important that the transfer is done in chunks. Instead, it would seem dynamic requests are better suited to the task as we can send the data byte by byte over a web socket connection and avoid the need to load the entire data set into memory. Our experiments lead us to conclude that on the client side it is not possible to save large amounts of data to a file that is received via a web socket without the use of some third party tool such as Flash or Java.

The solution that was chosen uses a combination of long polling to download the data and a web socket to provide real time progress updates; we call this configuration "tunnelling". Figure A.19 (on page 40) is a flow chart demonstrating how this system works

Long polling is a server push technology which enables the server and client to keep the connection open for an indefinite period of time [Bozdag et al., 2009]. Through the use of long polling, the server can send data as it is loaded and does not need to persist the data in memory, it can send it directly to the client. Long polling can also be used to support file downloads if the server configures the response header appropriately (see "Content-Disposition" header field in the HTTP 1.1 RFC [Fielding et al., 1999]); as a result the client browser can automatically write data to file as it is received, again avoiding the need to hold the entire data set in memory.

One problem with our current solution is how we determine the size that the file will be. If we do not specify the size of the file in our response then the user's browser will have no way of indicating how much of the download has completed; this is bad from a usability stand point. In our system the data will be coming directly from a database which stores all tweet and user data. Most database software has some method for counting number of records or calculating the storage required by a set of records; however, these are often inefficient and may not be available. In the case of retrieving the number of records, estimating the storage required by the records can be difficult if records vary in size. Tweets for example can vary in size significantly as some tweets are only a few characters long where others are hundreds.

In order to provide an accurate and visually appealing view of progress to the user it was decided that the use of web sockets should be combined with our long polling file download. A web socket is used to update progress bars displayed by the client's browser in real time, so that as we move through the search results, we move the progress bar proportionally. We do not need to know the precise storage size of the data set in order to achieve this, we simply need to know approximately the number of records. If the database has no efficient way of counting the number of records we can estimate progress by sorting results by timestamp. As we move closer to the current time we can move the progress bar proportionally.

Chapter 4

Implementation

4.1 Tools

4.1.1 Python

The project has been written in Python v2.7.3. Python is an agile programming language which is portable, easily extensible and has a wide range of third party packages [Chun, 2001]. It is particularly good for fast prototyping and development of a fast changing code base [Chun, 2001]. Our project development was fairly fluid with new ideas and changes to the design evolving over time, so it was particularly important that the language was flexible enough to support this.

4.1.2 Database

In the solution design section we explain how a database is used to persist data. We detail a number of requirements; most importantly we need to be able to read and write data in logarithmic time so that our binary tree structure (see section 3.3.3) can run efficiently. The database needs to be capable of handling a large amount of data without a significant reduction in performance as we may need to store many millions of tweets and users at any one time. Our system is both read and write heavy in that we have many tweets being written to the database every second (on high traffic streams) and many tweets being read from the database as multiple users download and view historical data.

For this project we decided to use MongoDB, a database which is not SQL based [Chodorow and Dirolf, 2010]. MongoDB has a number of advantages over SQL databases, first it has a dynamic schema which means that documents in the same collection (equivalent to rows in the same table of an SQL database) do not need to share the same fields or structure; in this way we do not need to spend so much time thinking about how to structure our database [Chodorow and Dirolf, 2010].

MongoDB is designed to be scaled out by distributing data across multiple MongoDB instances; this means that as load on the database increases we can easily scale to meet increased demands by adding more MongoDB instances [Chodorow and Dirolf, 2010]. The MongoDB community have investigated performance of MongoDB vs SQL based counterparts and found that MongoDB can perform significantly better in many scenarios, particularly where joins would normally be required [Kennedy, 2010]. High performance is a key design goal in the development of MongoDB and the consensus seems to be that for the majority of tasks MongoDB performs as well or better than SQL alternatives [Chodorow and Dirolf, 2010; Kennedy, 2010].

MongoDB supports indexing using B-trees; indexes improve database read performance by organizing data such that efficient lookups are possible [MongoDB, 2013a; Chodorow and Dirolf, 2010]. B-trees have $log_2(n)$ complexity for inserts and reads [Comer, 1979]. As a result we are able to fulfil our solution design requirement for logarithmic database performance.

A second requirement detailed in our solution design is for there to be some way of counting records so that we can provide progress bars which indicate progress through a download. MongoDB provides a count function which can be used for these purposes [Chodorow and Dirolf, 2010]. However, we found this to be too slow for our purposes and so switched to using the

timestamp of the data to indicate progress. To achieve this we index users and tweets by insertion time in timestamp form, and while iterating through a search space timestamps will gradually increase until reaching the end of the search space. In this way we can move progress bars proportionally to our position within the search space. While this may not be completely accurate as rate of inserts may increase or decrease over time, it does provide a cheap and reasonably effective method of keeping track of where we are in the download.

4.1.3 Python Packages

A number of third party packages were used in the development of this project; full details are listed in table B.7 (on page 53). Bottle is a micro web-framework for Python [Bottle, 2013]. We use this to provide static data such as HTML web pages to users. It allows us to use templates to mix Python code with HTML, JavaScript and Cascading Style Sheets [Bottle, 2013]. In our project HTML provides the basic skeleton of the display, JavaScript updates the display and provides other dynamic functionality and Python manages the back end data retrieval and analysis. Cascading Style Sheets are used as a central source for styling HTML objects e.g changing dimensions and colours [w3schools, 2013c].

Bottle meets our solution design criteria for our web server technology to provide a mechanism for caching static content (see section 3.4.2) because it has an in built caching mechanism which caches static content such as templates in memory [Bottle, 2013]. However, Bottle does not natively support web sockets or long-polling [Bottle, 2013]. This is where gevent and greenlet come in. The greenlet package provides us with objects known as "greenlets"; these are logical pseudo threads which run concurrently [Greenlet, 2013]. Unlike normal threads, there is virtually no overhead in creating and maintaining a greenlet; this means that we can run millions of greenlets simultaneously without hindering performance [Greenlet, 2013]. Gevent and gevent-websocket packages allow us to leverage this in a bottle web server such that each connection runs in its own greenlet and thus enables us to maintain many web socket or long polling connections concurrently [Bottle, 2013]. We use a release candidate of gevent because we experienced problems with the current official release in its interaction with bottle; upgrading solved these problems.

In order to connect to external HTTP APIs such as the Twitter streaming API, the Google geocoding API and the OpenMapQuest geocoding API we need some way of connecting to HTTP end points. We use Python requests package to do this. It supports keep alive connections which are necessary to maintain a persistent connection to the Twitter streaming API and is well documented [Reitz, 2013]. We need authorisation to use a user's Twitter account with Twitter APIs; to do this we implemented 3-legged oauth with the help of packages requests-oauth and oauthlib [Twitter, 2013a]. These packages allow us to sign requests using HMAC-SHA1 as required by Twitter APIs [OAuthLib, 2013; Twitter, 2013a].

Lastly, we use PyMongo to communicate with MongoDB. PyMongo is a simple wrapper around MongoDB functionality which enables Python programs to interact with it [PyMongo, 2013].

4.1.4 JavaScript and Style Sheet Plugins

A number of third party JavaScript and Cascading Style Sheet (CSS) based plugins were used in the development of the web front end; for exact details of these see table B.7 (on page 53).

jQuery greatly extends native JavaScript functionality, providing a range of general purpose browser portable functions [w3schools, 2013a]. It is used extensively throughout the project, particularly when manipulating HTML DOM objects.

Twitter bootstrap is used for the positioning of items on the page and provides a mixture of CSS and JavaScript entities. It provides fluid layouts which resize and reposition objects as the window changes in size; in this way support for tablet and mobile devices is facilitated

by streamlining the interface at smaller resolutions [Bootstrap, 2013]. Twitter bootstrap also provides a number of useful display objects such as the collapsible accordion which is used to display help in parts of the system [Bootstrap, 2013]. When clients click on relevant entities the help accordion shows corresponding information about the entity which the user is interacting with. One of the main advantages of Twitter bootstrap is browser portability; layouts created with bootstrap will look the same across most modern browsers [Bootstrap, 2013].

Leaflet is used to display maps. Leaflet is open source, works well on mobile devices, is well documented and is easy to use from both a human computer interaction (HCI) stand point and from a programming perspective [Leaflet, 2013]. We use cloud made to provide tile images used in the display of our maps. Leaflet maps are not tied down to a specific tile provider but cloud made was chosen for its reliability (99.9% up time) and wide range of map styles [Leaflet, 2013; CloudMade, 2013]. Note that cloud made is simply an end point which we point leaflet to, it is not a code based plugin.

We also use a leaflet plugin called "Leaflet.markercluster" [MarkerCluster, 2013]. This is used extensively for displaying location and influence data. It groups nearby markers together so that while zoomed out we see 1 or 2 markers and when zooming in the markers split up into smaller units [MarkerCluster, 2013]. In this way we can mark thousands of items on a map without running into performance issues or making the map difficult to understand with too much information. We also use this for displaying percentages, it is useful to display influence data in this way; see figure A.24 (on page 43). As we zoom in to percentages more nodes appear and we see more granular detail as to which locations make up that aggregate percentage.

A number of other plugins which are based on jQuery are used. "jQuery UI" is a plugin which provides a number of advanced user interaction objects; we use it purely for its slider bar object to allow users to select time periods and batch sizes (see section 4.2 for full details on time periods and batch sizes). "jquery-cookie" is used to add and remove cookies from client browsers. "jquery-custom-scrollbar" and "jquery-mousewheel" are used to style scrollbars.

4.2 System

Throughout the report so far we have discussed individual components or aspects of the system such as: twitter streams, geocoding, influence, follower enrichment and interaction with end users. This part of the report will detail how these components interact with each other to produce an end product and what the overall functionality of this product is.

4.2.1 Overview

Clients are introduced to the concept of a "search stream". This is a client defined search on a set of keywords or geographical locations with which tweets are retrieved and influence analysed. Clients can search for disjoint keywords i.e retrieve tweets which contain at least one of word A or word B, and can also search for joint keywords i.e retrieve tweets which contain both word A and word B regardless of word order. Clients can select areas on the map by searching for locations by name, or by drawing areas on the map using their mouse. When selecting areas, one of three selections can be made, each represented by a different colour. Green represents an area which the search stream should retrieve Twitter geotagged tweets from. Red represents an area to use as an influence source. Yellow represents an area to receive Twitter geotagged tweets and use as an influence source. See figure A.20 (on page 41) for an annotated example. The system will pick a selection of tweets which were created by users which are associated with an influence source chosen by the client. When tweets of an influence source are selected, users which created the tweets go through the follower enrichment process. While enriching, spatial influence data is gathered and associated with both the user and the location associated with that user.

Clients can access data on a per location basis at a city, country and continent level. We plot cities, countries and continents on a map as we receive tweets associated with them. This map is the entry point for search streams and upon clicking on a location, clients are directed to a new window with information about that location.

In this location page, users can see live and historical data by alternating between tabs. Live data is updated via a web socket and contains a limited number of the most recent tweets associated with that location, with the most recent closest to the top. On the server side a small number are cached so that on low traffic locations there are always some tweets displayed to newly connecting clients. On the client side there is a fixed limit to the number that will be displayed to avoid overloading the browser; once this limit is reached the oldest tweets are discarded.

Historical data is updated via AJAX HTTP requests which download data from the server. AJAX requests are simply HTTP requests initiated by JavaScript code which can be used to asynchronously update parts of the page without reloading the entire page [w3schools, 2013b]. Clients can scroll between time periods using a slider bar with a start and end time. The first 20 tweets are shown but clients can scroll down and more will automatically be loaded. On the server side tweets are retrieved via a MongoDB database query. Influence of the location is also displayed for the specified time period. Influence is displayed in the form of a list of target locations which it has influence on. The list contains details of how many followers are located in the target location as a number and as a percentage. The list is split into cities, countries and continents and also includes a map visually showing percentages. See figure A.23 and A.24 (on page 43) for screenshots of this.

Information about Twitter users is also available on a per user basis with information such as number of followers, user name, user description and user location. The user page is accessible from the location page by clicking on a tweet; a new page is opened with details of the user who created the tweet. One interesting feature is the way in which follower information is displayed. Clients can switch between user name and display picture view. In user name view the user names of followers are displayed and the client can hover their mouse over followers and a small popup will appear with the follower's display picture (see figure A.22 on page 42). When the display picture view is enabled the opposite is true, display pictures are displayed and the client can move their mouse over a picture to view the follower's user name (see figure A.21).

Clients have the ability to download data for specific locations or for the entire search stream. Three download types exist, the first is for downloading tweet and user information; two files will be downloaded in this case, one for tweets and one for users. The user file contains information about the users which created the tweets. Second, users can download "full follower information" which contains details of users for which we have enriched follower information, and details of their followers. Third, users can download "short follower information" which contains details of users for which we have enriched follower information but excludes details of their followers to save space in the download. The download is in the form of a CSV file and contains all details we have about the users and tweets within the search space including geocoding, influence data and Twitter raw data. See figure A.25 (on page 44) for a screenshot of the download setup page. Downloads are split into batches, and users can select a batch size before starting the download. By limiting the batch size users can download partial data sets, helping to avoid long waits while many gigabytes worth of data are downloaded. After each batch the user can continue and the download will start up again creating new files to store the next batch of data. While downloading, two progress bars show the progress through the current batch and the progress through the total download as shown by figure A.26 (on page 44).

4.2.2 User Interface

This section aims to describe the design principals that were followed during the development of the system, and how this resulted in the use of a wide variety of user interface elements throughout the system.

Brown describes a number of generic design principals for the development of user interfaces [Brown, 1998]. He highlights a wide range of design principles of which four were of particular relevance for the purposes of this project [Brown, 1998]:

- Enforce interface consistency by representing repeated objects continuously throughout the system.
- Minimize amount of cognitive processing required to understand data.
- Minimize need for user to enter raw data.
- Avoid overcrowded or cluttered displays.

The system needs to display a large amount of geographical based data; tweets, users and influence are grouped by continents, countries and cities. To display this data we use maps so that users do not need to manually determine where in the world a location is situated. The maps are generated using JavaScript plugins leaflet and "Leaflet.markercluster" as detailed in section 4.1.4. Influence statistics are more easily visualised in the form of a map as clients can compare percentages visually. Colour coding is used to highlight highly concentrated nodes; green represents weaker nodes and red represents stronger nodes. Figure A.24 shows a screenshot of the interface. We can clearly see that the entity has strong influence on the United Kingdom as shown by the displayed percentage of 63.7% and the orange colour coding. The map groups nodes together and shows an aggregate percentage when zoomed far enough out, which helps to reduce clutter and overcrowding. If a client is interested in precise influence values, we also provide a text based display where locations are listed with numerical and percentage based results; figure A.23 shows a screenshot of this interface. Clients can click on a location name to view that location on the map; the map will automatically reposition and zoom to the required location. This simple method of switching between the text and map display reduces the cognitive processing required by the user.

Throughout the project we make extensive use of tabs to avoid overcrowding and cluttering the display. The Bootstrap plugin provides us with the tabs functionality as described in section 4.1.4. Figure A.24 shows an example of how we use this functionality. Tabs allow us to split the view into three distinct independent parts shown as "Tweets", "Influence" and "Download Data" in figure A.24. Nested tabs further split the interface into distinct related sub sections. In figure A.24 the map object persists when switching between cities, countries and continents but the nodes displayed on the map change. In this way we avoid overcrowding the display by separating out each location type but maintain interface consistency by reusing the map to display each location type individually depending on which tab is selected. This has the important benefit that changes in map position and zoom persist between the city, country and continent display. This reduces cognitive processing required because clients do not need to interact with three separate maps; instead clients interact with one single map and only need to consider the important changes in data, i.e. the nodes displayed on the map.

On the user screen there is an intuitive view of a user's followers. This view enables clients to see a user's followers in a visually appealing form by viewing user names or profile pictures as displayed in figure A.21 and A.22. This view is custom designed for the purposes of this project and uses only basic features of Bootstrap to facilitate the tooltip popup and the fluid layout which enables the display to react to change in window size. This display is particularly good at engaging with clients and turns an otherwise relatively bland list of user names into a visually appealing selection of display pictures. More over, it allows clients to get a feel for the type of followers following a user through observing the display pictures.

Figure A.21 and A.22 highlight another core interface feature utilized in the development of this project. Users may have many thousands of followers, so it is often infeasible to display all of the followers of a user at once. At the same time, a client might be interested in looking at a large number of followers for a particular user and so sometimes it may be necessary to display a large number in order to meet the needs of a client. The solution that was implemented is known as "infinite scrolling". When a client scrolls to the bottom of the list of followers more will be loaded. This process continues until all followers have been loaded or the client stops scrolling. In this way we can reduce load on the server and the client's browser by sending only a subset of the followers of a user, but if the user attempts to view more followers than currently loaded then more are displayed automatically. This is an intuitive design which provides clients with as much data as they need, but does not require the client to input this requirement as raw data. The display reacts to the needs of the client automatically and as a result minimizes cognitive strain on the client.

A final critical interface component used in this system is the use of "breadcrumbs" based navigation. This is a system of navigation in which clients have a position within the system and can navigate backwards using breadcrumbs [Gube, 2009]. For example, when viewing the location page for the city of London in the United Kingdom, the breadcrumbs navigation is shown by figure A.27 (on page 44). This title bar is shown on all pages in order to maintain interface consistency and to ensure that the the user only needs to remember one core method of navigating back through the system. Clients know that they are on the London page because the London item has an inset style. London is a part of the United Kingdom and Europe and so users can traverse backwards into these parent entities. The overall search stream can be reached through the "Search Stream" button and the landing page for the system as a whole can be reached through clicking the GeoTweetSearch logo which is present on every page.

4.2.3 Architecture

Figure A.28 (on page 45) shows a very high level view of how the modules of this system fit together. Tweets are received via the Twitter Streaming API and handled by the Twitter stream module; they are then geocoded by the geocoding module. A number of these are processed by the follower enrichment module which loads information about followers of the users which created the tweets; the rest are passed through. Lastly the web front end interacts with this data and displays it to client browsers.

Figure A.29 (on page 45) shows how this system was implemented at a threading level. A thread pool is a collection of persistent threads which perform a task [Gift, 2008]. In this system thread pools share a single input queue and write to one or more output queues. In this way for example, we can geocode two items from the database concurrently; this is likely to be faster than doing so sequentially if the database can handle multiple requests in parallel. Thread pools and queues are an efficient method of sharing data and tasks between threads in Python [Gift, 2008]. In a gevent based system threads are replaced by greenlets which are cheap to create [gevent, 2013]. Despite this, there are still advantages to avoiding thread creation by using thread pools; namely, latency is reduced as threads can straight away begin processing the next item in the queue without any intermediate step. One problem with a queue based system is the potential for queues to fill up and cause memory exhaustion. To prevent this threads automatically drop resulting data if output queues become too full. While this is not ideal, it is making the best of a bad situation.

In figure A.28 there are two search streams running (the concept of a search stream is explained in section 4.2.1). For the life time of the search stream two threads are created, a twitter streaming thread and a follower enrichment thread. This is because in the Twitter API each user account can have one search stream open at a time, and rate limits associated with follower enrichment are on a per account basis [Twitter, 2013e,g,h,b]. Tweets and user data generated by these two threads are associated with the account that setup the stream, so that

data from different search streams can be differentiated when outputting data to clients.

After a tweet is received from the streaming API it is passed onto the geocoding module, the entry point being the "geocode from memory" thread pool where tweets are geocoded from the in memory cache. Unsuccessfully geocoded tweets are passed to the "geocode from cache" thread pool which attempts to geocode using the database. Tweets which are unsuccessfully geocoded here are passed to the "geocode from external" thread where they are geocoded by the online geocoder. Note that geocoding from online geocoders is rate limited so there is no sense in using a thread pool as we can easily meet the rate limit with one single thread. Successfully geocoded tweets at any stage in the flow are passed directly into the follower enrichment module, the entry point being the "follower enrichment gate" thread pool. Here a selection of tweets are passed to their corresponding "Twitter follower enrichment" thread; this occurs only when there is space in the thread's queue. The rest are simply passed through. The "Twitter follower enrichment" thread retrieves follower information and performs influence analysis. All resulting data is passed to the output thread pool which writes data to the database and passes updates into the "data provider tree"; see section 3.4.2 for full details on the roll of this tree structure. The display thread signals updates in the tree to web sockets in real time. When clients initially connect to our front end (built using the bottle Python package) via a web socket they may receive a cached copy of data from the data provider tree. Lastly, historical data may be requested which is retrieved directly from the database.

4.3 Testing

Our main method of testing involves extensive logging. Not only do we log erroneous situations but we also log performance statistics showing for example, how many tweets we are dropping due to high load. Using Linux scripts we can extract relevant information and build graphs showing system performance at a module and at an overall level. For example figure A.7 (on page 32) shows overall geocoding success and failure rates across the memory, cache and external thread pools over a 70 hour period. Another example is figure A.5 which shows hourly success and failure throughput of the "geocode from external" thread. We can use this to verify that we are properly rate limiting use of the Google online geocoder to 2500 requests a day (approximately 104 an hour, once every 35 seconds). Many different graphs showing performance of different parts of the system can be generated, and in fact all graphs in this report are generated in this way by running a Linux script on log files to retrieve statistics and using a spreadsheet package to convert the raw data into graphs.

Chapter 5

Evaluation

5.1 Geocoding

Figure A.7 (on page 32) shows overall geocoding success and failure rate over a 70 hour period. We notice that this is fluctuating with between 20% and 30% success rate which is lower than rates of other research e.g. Lima and Musolesi found approximately 36% of Twitter users in their data set could be geocoded at town level [Lima and Musolesi, 2012]. Takhteyev et al. found 57% of Twitter users in their data set had a named location which could be geocoded [Takhteyev et al., 2012]. There are two likely reasons for these differences. First, we are very specific in what we consider a successful geocode. We only consider users which we are able to geocode to cities or large towns, we ignore smaller towns and street or postal code level addresses. Secondly, unlike the research performed by Lima and Musolesi and Takhteyev et al. we cannot use the online geocoder for every single user, and we cannot manually geocode results to improve accuracy for users with malformed or otherwise non standard location fields [Lima and Musolesi, 2012; Takhteyev et al., 2012]. We can not do this because the system is a real time system and aims to consume as much of the feed as possible; geocoding is rate limited and so it is impossible to keep up with the Twitter streaming API unless we use some form of cache.

We have highlighted a number of limitations in the geocoding module. First, we may want to consider other location types such as small towns, villages, street addresses, points of interest or other geographical location types beyond cities. Second, the requirement for a user to be associated with a city or large town means that we do not consider users which we could directly geocode to a country or continent, for example if their location field is “France”. Whilst we do classify users into countries and continents, we do not consider users which are not associated with a city, even if we could geocode them to a country or continent. For example, we can associate a user with “London, United Kingdom, Europe” but we cannot associate a user with “United Kingdom, Europe”. This is a fairly major limitation as research by Lima and Musolesi suggests that as much as 7% of users may be identifiable only by country [Lima and Musolesi, 2012]. This limitation simplifies implementation but should be removed with further development of the project. A third course of action could be to investigate the performance of other geocoding APIs. The system currently supports OpenMapQuest and Google geocoding APIs; all analysis in this report was done using the Google geocoding API. It is very easy to setup online geocoders with the geocoding module of the system, so we could easily switch over to a new online geocoder. We currently support the running of only one geocoder at a time, but we could combine geocoders to increase accuracy, for example Google may fail to geocode something which OpenMapQuest can geocode.

A further problem is that a large proportion of users have no location information associated with their account or their tweets. In our analysis we found that over a 70 hour period on average 39.37% of users encountered had no location information (see figure A.33 on page 48). As a result our solution has no way of geocoding their location and cannot use their data for any geographical based analysis. This is a limitation of the social network but it is hoped that Twitter will encourage users to provide location data as the social network matures. There are a number of potential solutions which could mitigate the problem. First we could expand our system to consume data from other social networks which place a greater emphasis on

geographical data e.g. Foursquare is a social network which personalises content based on the location of users [Foursquare, 2013]. Second, instead of geocoding the location field, we could implement a geocoding strategy which analyses the contents of tweets. Hecht et al. implemented a machine learning technique to geocode Twitter users at country level by analysing the contents of their tweets [Hecht et al., 2011]. While a machine learning technique may increase the number of users that we can process, it may be difficult to maintain the level of precision and accuracy that we need in order to analyse geocode data at a city level. For future development perhaps the online geocoding techniques in our solution could be augmented by some form of tweet contents analysis in order to improve overall accuracy and success rate.

Our geocoding module achieves fairly consistent overall success rates of between 20% and 30% while processing around 50 tweets per second (see figure A.1 and A.7). It caches data in memory to minimize load on the database and solves the issue of rate limited online geocoding; table B.3 shows how in normal run time 99.84% of the load is consumed by the in memory cache. While there is work to be done on improving success rates, the framework developed for geocoding is valuable as it enables real time geocoding with high throughput.

5.2 Performance

We performed extensive analysis of the system while it was subscribed to a high traffic stream. We wanted to see what load it could handle and if any bottlenecks in performance could be found.

Figure A.31 shows that performance begins with around 25,000 tweets being processed every hour; this is 7 per second. At around 8pm on the 5th of April performance falls to approximately 10,000 per hour, around 3 per second. This is a substantial drop in performance, so we need to understand what has triggered this. Our system writes all tweets and user data that it receives to the database in real time, if our database is slow to insert data then performance of the overall system will be hindered. Figure A.32 shows what is happening in our database. We have 1.3 million users and 900,000 tweets stored in our database at the time of the drop in performance. In total, this constitutes a total index size of around 475 megabytes. Indexes improve database read performance and are necessary when dealing with millions of records [MongoDB, 2013a]. Inserts slow down if MongoDB can't fully load indexes into memory [MongoDB, 2013c]. The database server used in our analysis is running on a machine with 1 gigabyte of memory and so it is likely that at the time of the performance drop MongoDB reaches a memory limit which triggers performance issues and slower inserts. We have identified the source of the performance issues as being the configuration that we used while running our analysis, namely the MongoDB server. We can easily allocate more memory to the MongoDB instance to tackle the drop in throughput from 7 to 3 tweets per second.

However, Figure A.30 shows that before the drop in performance we were still unable to fully process all successfully geocoded tweets. Up to around 5% of the Twitter stream was unprocessed as a result. The remaining throughput issue of moving beyond 7 tweets per second can be tackled by "scaling out" our MongoDB instance into multiple shards [MongoDB, 2013b]. Note that while performing our analysis we were running with a single unsharded MongoDB instance. By changing to a sharded setup we can distribute the load between multiple machines and perform multiple writes in parallel [MongoDB, 2013b]. Our thread pool implementation is already prepared for this setup and we can simply add more threads to the thread pool if we decide to move to a sharded setup.

Whilst we were unable to fully consume the high traffic stream which we performed our analysis against, it is my opinion that 7 tweets per second being inserted into a collection with over 1 million records is not bad performance wise, especially when you consider that performance up until the drop was fairly consistent. We have identified the bottleneck in our analysis and it is something which can be solved without changes to the design of our system;

database hardware configuration changes should increase throughput. If the system is to be used for high throughput analysis we recommend setting up a sharded environment to spread the load on the database between multiple machines.

5.3 Influence

We implemented a measure of influence known as "spatial degree centrality" [Lima and Musolesi, 2012]. While it is a valid measure of influence, it has its limitations [Lima and Musolesi, 2012].

First, a user with lots of followers will have more influence than a user with very few followers. This means that very popular nodes e.g. news agencies or celebrities will have a disproportionate impact on results, which may not be the desired behaviour [Lima and Musolesi, 2012]. To counter this Lima and Musolesi introduce other measures of influence such as "spatial degree ratio" which normalizes number of followers by describing influence as the *ratio* of neighbours inside the target area rather than the *number* of neighbours [Lima and Musolesi, 2012]. Second, we are only analysing the potential influence of users by analysing their neighbourhood structure [Lima and Musolesi, 2012]. There are many more areas which could be explored, for example we could look at the actual real time influence of users by analysing retweets. The system was built with future development in mind and so it should be relatively straightforward to modify the code base to support other measures of influence.

Table B.4 shows performance of the follower enrichment module over a 70 hour period. We see that on average 30 users an hour are processed and the data of 16,671 followers analysed. Over a 24 hour period this amounts to over 400,000 followers and 720 followees[1]. For many streams 30 users an hour will not be enough to fully consume an influence source; a significant number of users may be passed through without being analysed. The system prevents the number of users waiting to be analysed from exceeding a preconfigured size and populates the queue only when there is space available within this limit. In this way we randomly select users to process every few minutes and so gain a sample which is to some degree representative of the stream. Sample size does impact the extent to which data and analysis is representative of the stream; determining the ideal sample size is a common problem in many fields [Eng, 2003]. In future development of this project we anticipate a need for statistical analysis to help clients decide whether the data behind influence results is representative enough of the search space to provide valid and reliable results. In general, low traffic streams and influence sources will be better represented because less traffic will pass through without analysis. Similarly, a search space which covers large periods of time will also be more representative as a larger sample will be analysed.

5.4 User Interface

Throughout development considerable emphasis was placed on developing a user friendly and intuitive user interface. We used maps to display location based statistics and we developed our own method of displaying followers using a grid of display pictures and user names. Breadcrumbs are used to ensure that clients are always able to identify their location within the system and to enable clients to navigate backwards to parent pages. Tabs are used extensively to partition information and to reduce clutter and overcrowding. A combination of clever use of Bootstrap, jQuery UI and Leaflet map objects mixed with the development of our own bespoke objects such as the user follower display has lead to a slick and highly usable system. Future development may see our bespoke follower display system reused to display followers at a city, country or continent level or for other purposes such as consuming Twitter vine feeds; Twitter vine is a social network in which 6 second video clips are posted [Twitter, 2013i].

[1] A followee is a user with a neighbourhood of followers, a follower is a user which is in the neighbourhood of a followee.

Chapter 6

Conclusion

We have successfully developed a system to retrieve, process and analyse large quantities of social network data in real time. The end product allows clients to consume Twitter streams and generate influence analysis of users submitting tweets in these streams. This system allows users to view data in an aesthetically pleasing format with for example the use of maps (see figure A.24). In addition users can download large quantities of data produced by search streams and can do so for specific time periods and locations.

The system handles approximately 180,000 tweets an hour from the Twitter streaming API (see figure A.1) and is resilient enough to handle spikes in load. Much emphasis has been placed on optimizing performance so as to maximize throughput and deal with vast amounts of historical data in the form of millions of Twitter tweets and users. Four substantial sub systems have been developed including: Twitter data retrieval, geocoding, influence analysis and a web based front end. Through the development of each individual sub system a number of unique and challenging problems were encountered which warranted the use of advanced data structures and models e.g. binary trees, publish/subscribe model.

Performance of each sub system and the system as a whole has been thoroughly evaluated and quantified. The system performs reasonably well but limitations have been identified which future development may look to alleviate. In particular, the success rate of the geocoding module could be boosted by considering locations other than cities and large towns. Our analysis has shown that our system is heavily dependent on the performance of the MongoDB instance that it runs against. As a result, it is recommended that care is taken to ensure that the database configuration is sufficient for the intended use of the system.

The main limitation of the project is that it implements a single measure of influence which does not differentiate between normal users and very popular users such as news agencies. In addition, it only looks at structural influence, ignoring factors such as retweets. The project was developed with future development in mind and we expect it to be reasonably straightforward to reuse the existing code base to implement other measures of influence.

Appendix A

Diagrams

A.1 Figure 1

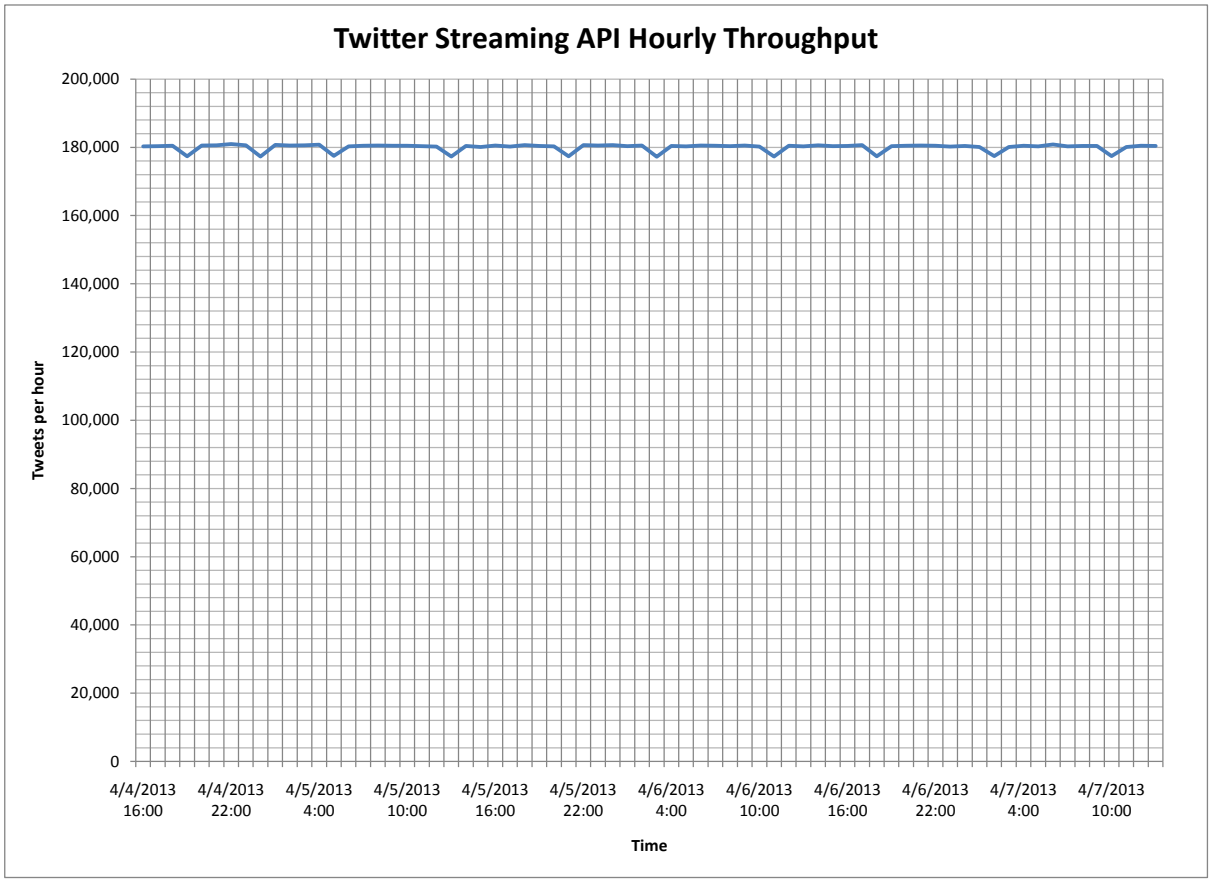


Figure A.1: Graph showing an average throughput of 180,000 tweets per hour over a 70 hour period subscribed to a high traffic stream via the Twitter streaming API.

A.2 Figure 2

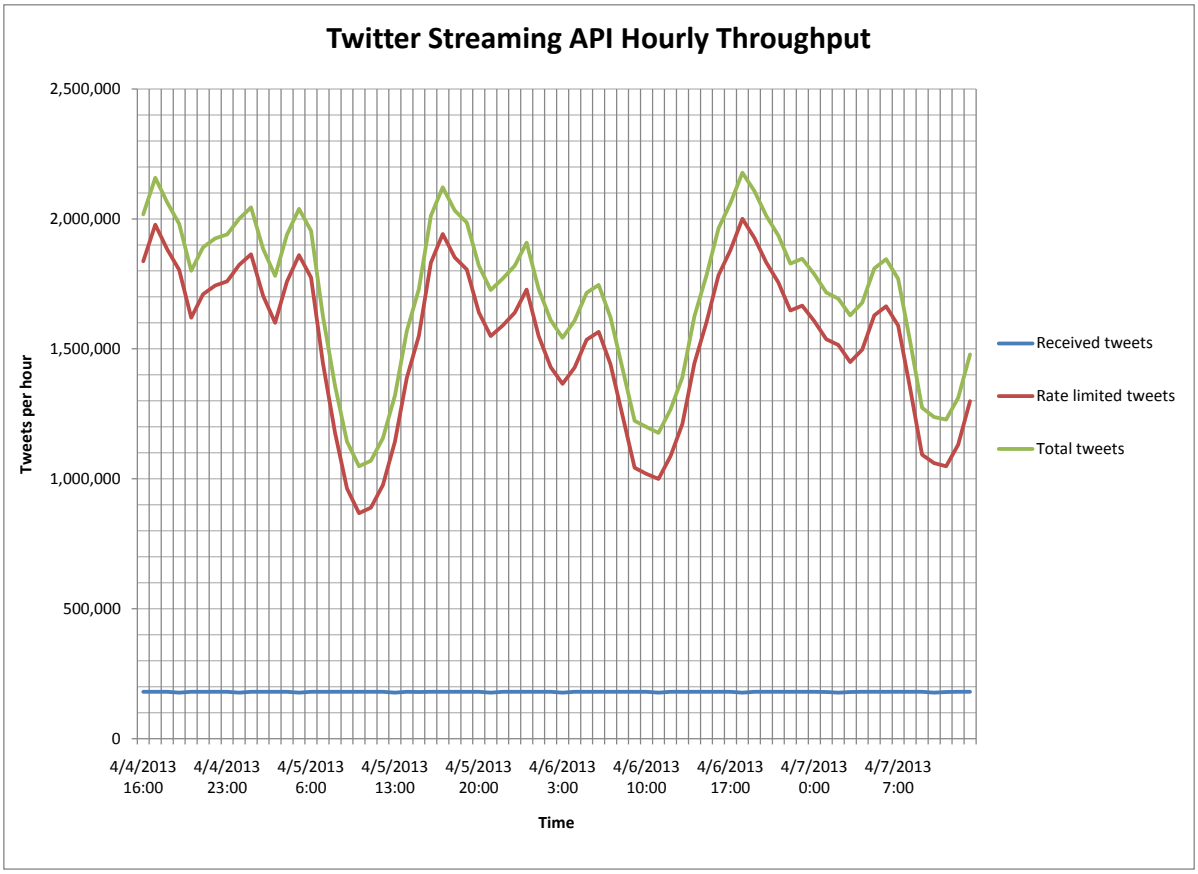


Figure A.2: Graph showing throughput of the Twitter streaming API, showing both rate limited and received tweets. Rate limited tweets are tweets that were in the stream but not sent to our application because of the rate limit. Received tweets are tweets which our application actually received through the API. The graph was built using data gathered over a 70 hour period subscribed to a high traffic stream.

A.3 Figure 3

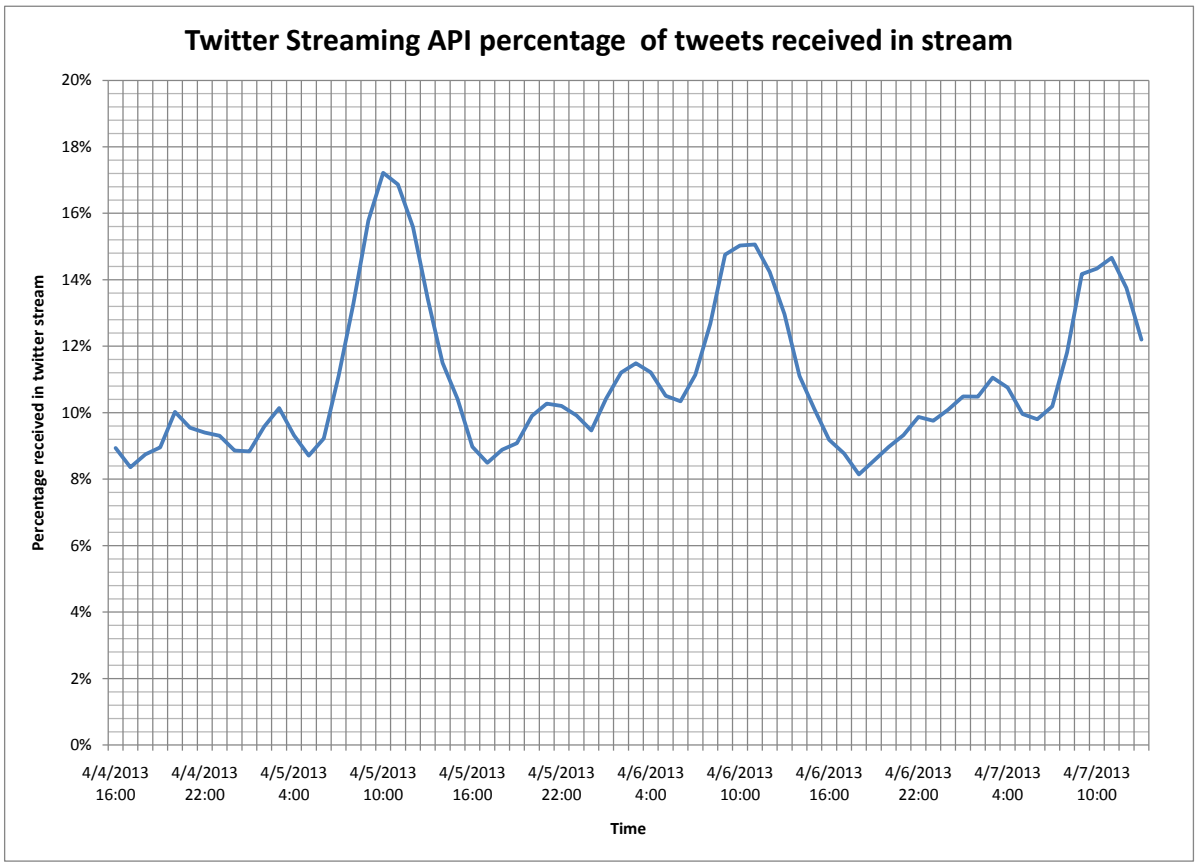


Figure A.3: Graph showing that we received between 8% and 18% each hour of a high traffic Twitter stream over a 70 hour period. The remainder of the stream was rate limited by the Twitter streaming API.

A.4 Figure 4

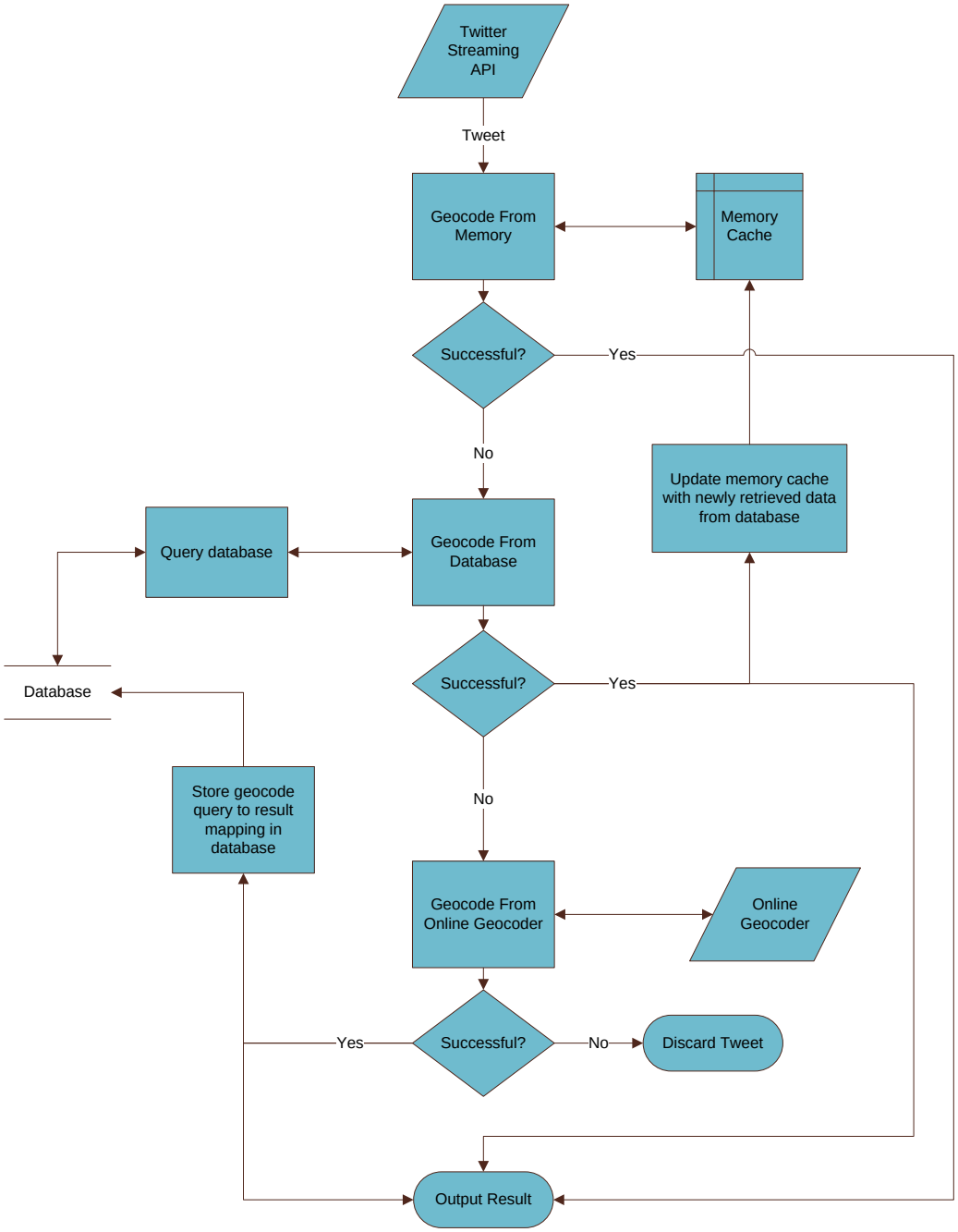


Figure A.4: Flow chart showing the geocoding process for geocoding the user provided location field associated with user accounts.

A.5 Figure 5

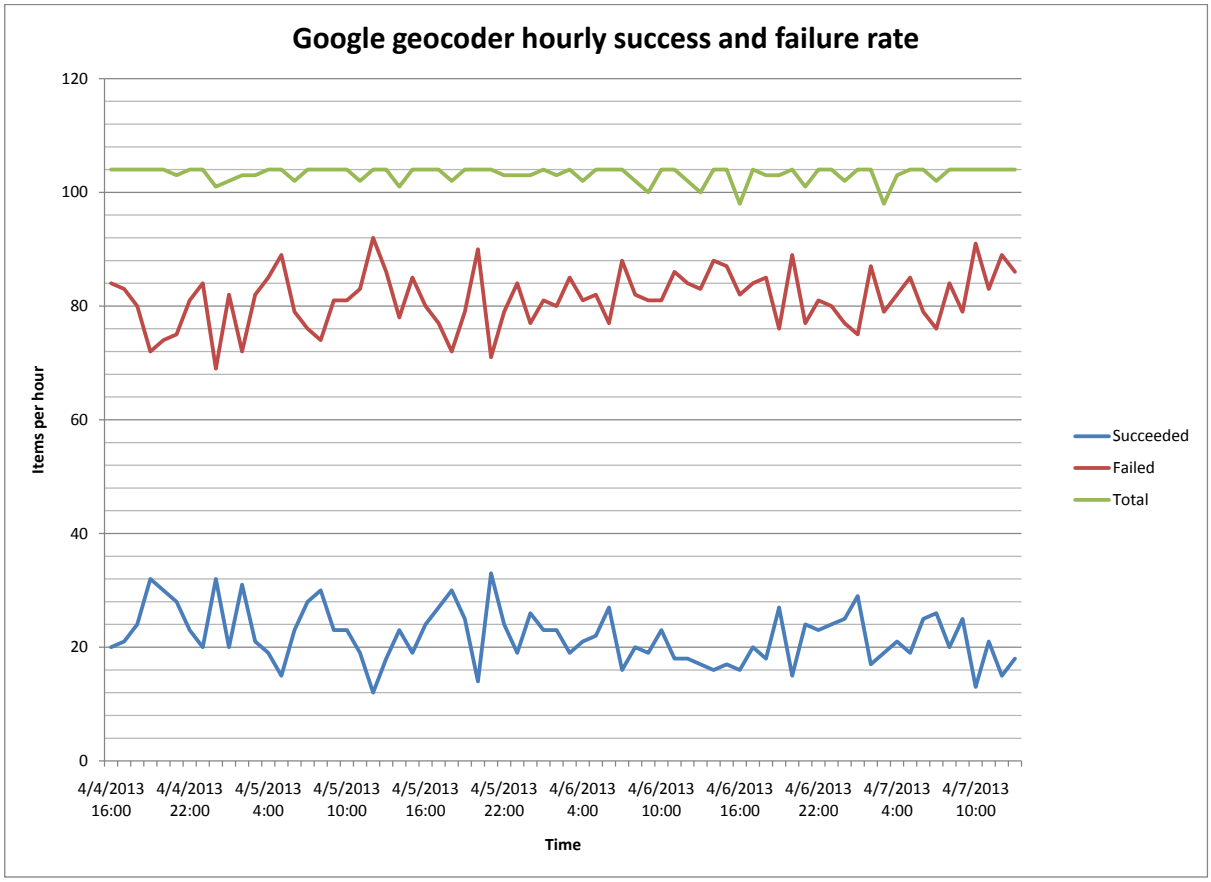


Figure A.5: Graph showing the hourly success and failure rate of the Google geocoding API to geocode user location fields from users who submitted tweets retrieved by the Twitter streaming API over a 70 hour period.

A.6 Figure 6

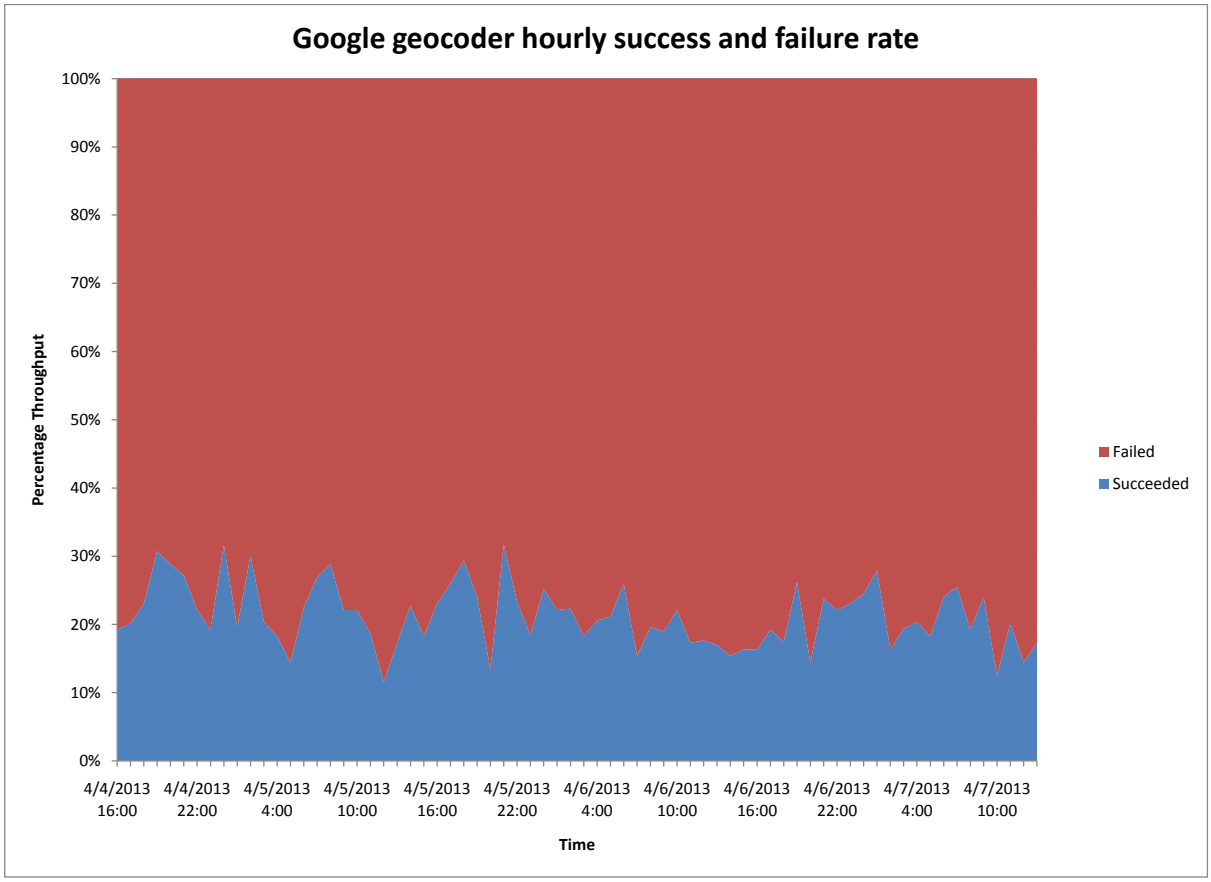


Figure A.6: Graph showing the percentage hourly success and failure rate of the Google geocoding API to geocode user location fields from users who submitted tweets retrieved by the Twitter streaming API over a 70 hour period.

A.7 Figure 7

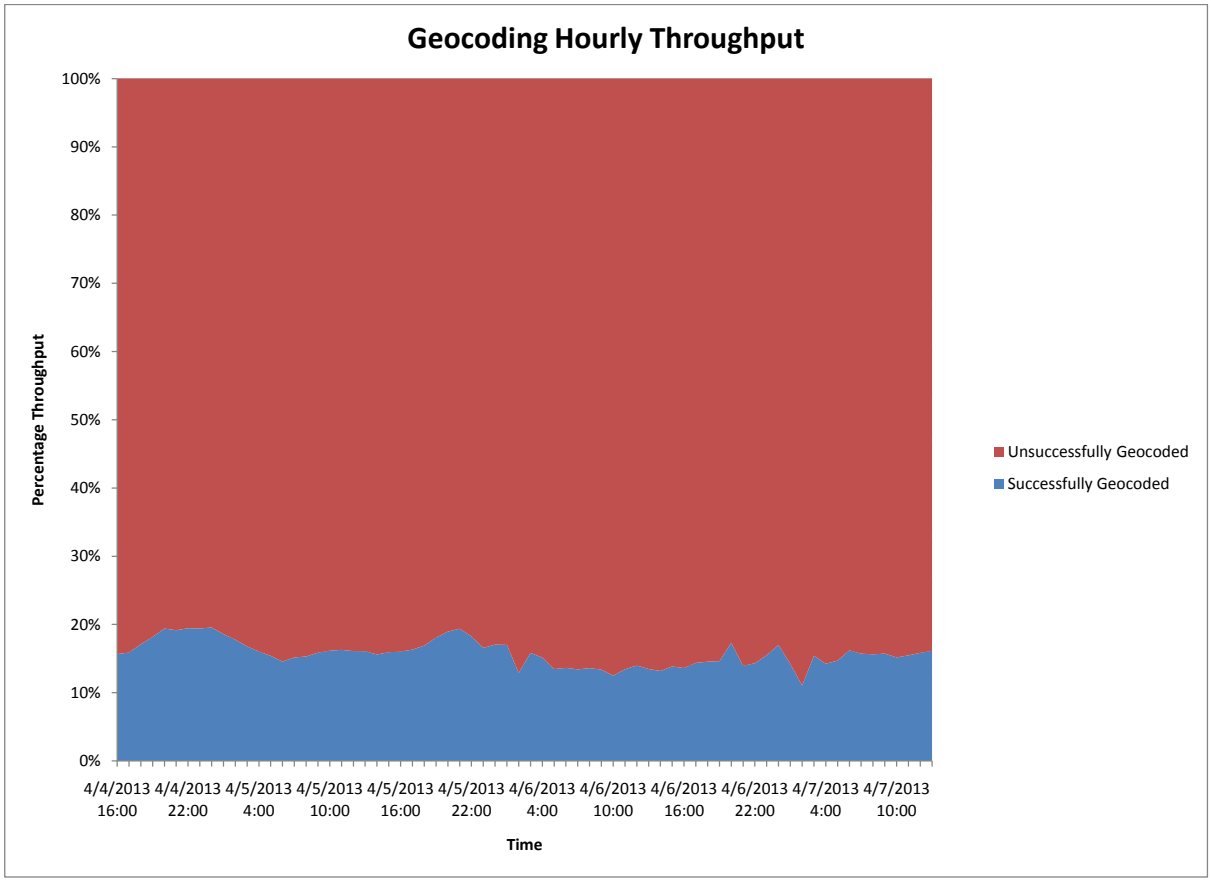


Figure A.7: Graph showing the overall percentage hourly success and failure rate of the entire geocoding process while geocoding the user location field of users who submitted tweets retrieved by the Twitter streaming API over a 70 hour period.

A.8 Figure 8

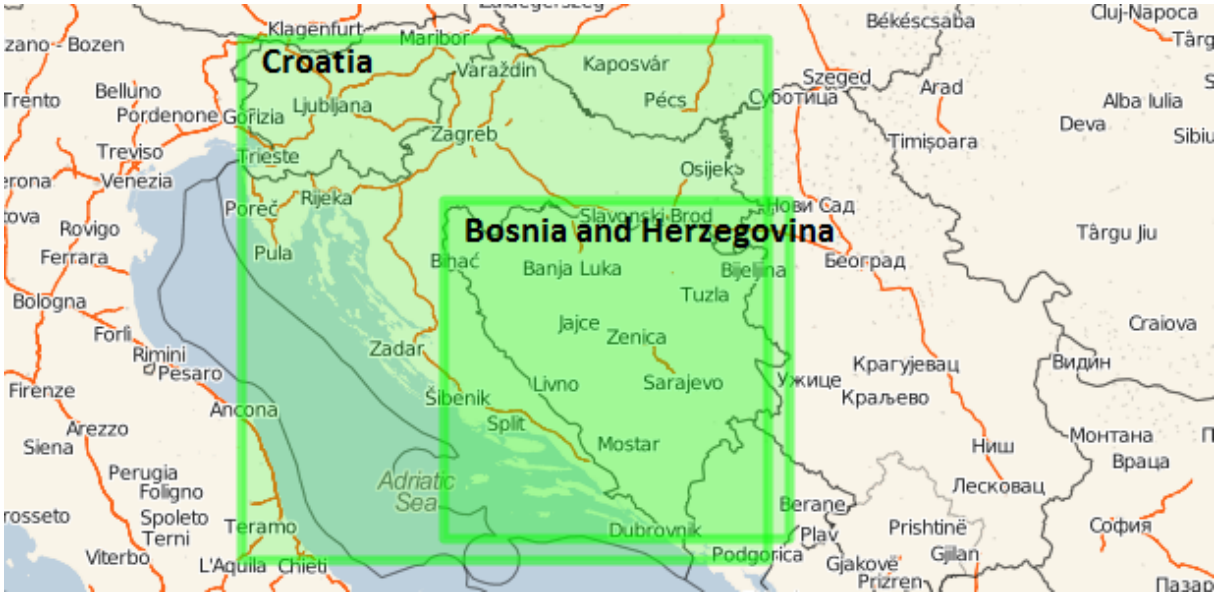


Figure A.8: Map showing bounding box of Croatia and Bosnia overlapping.

A.9 Figure 9

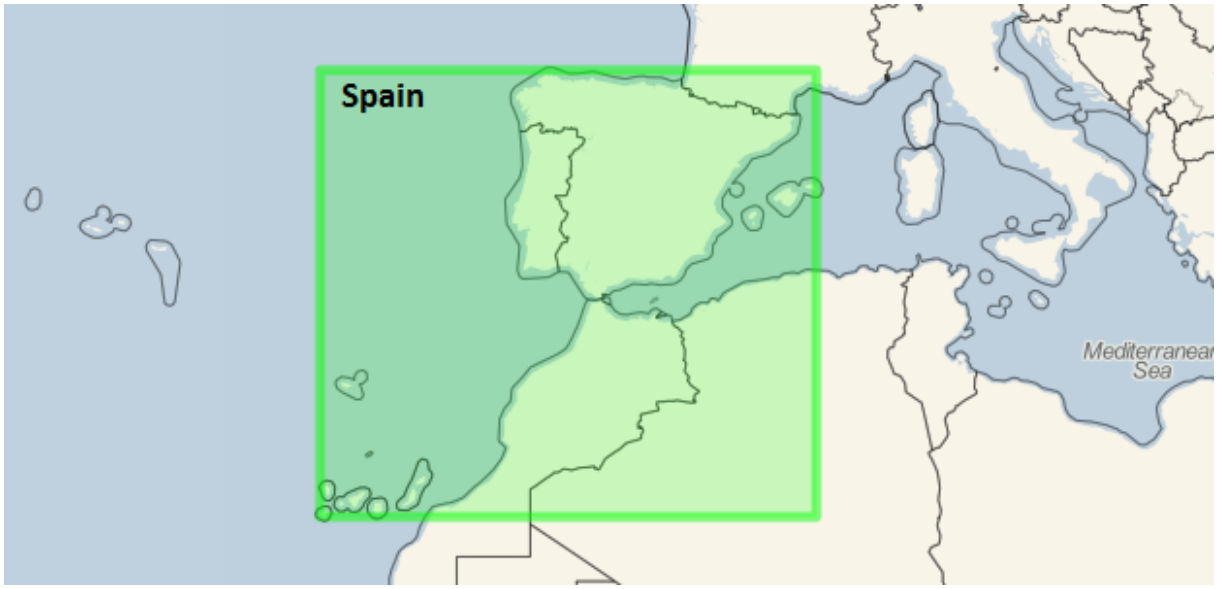


Figure A.9: Map showing bounding box of Spain which includes the Canary Islands (bottom left). As a result of the Canary Islands, the bounding box stretches over Portugal and parts of Morocco.

A.10 Figure 10

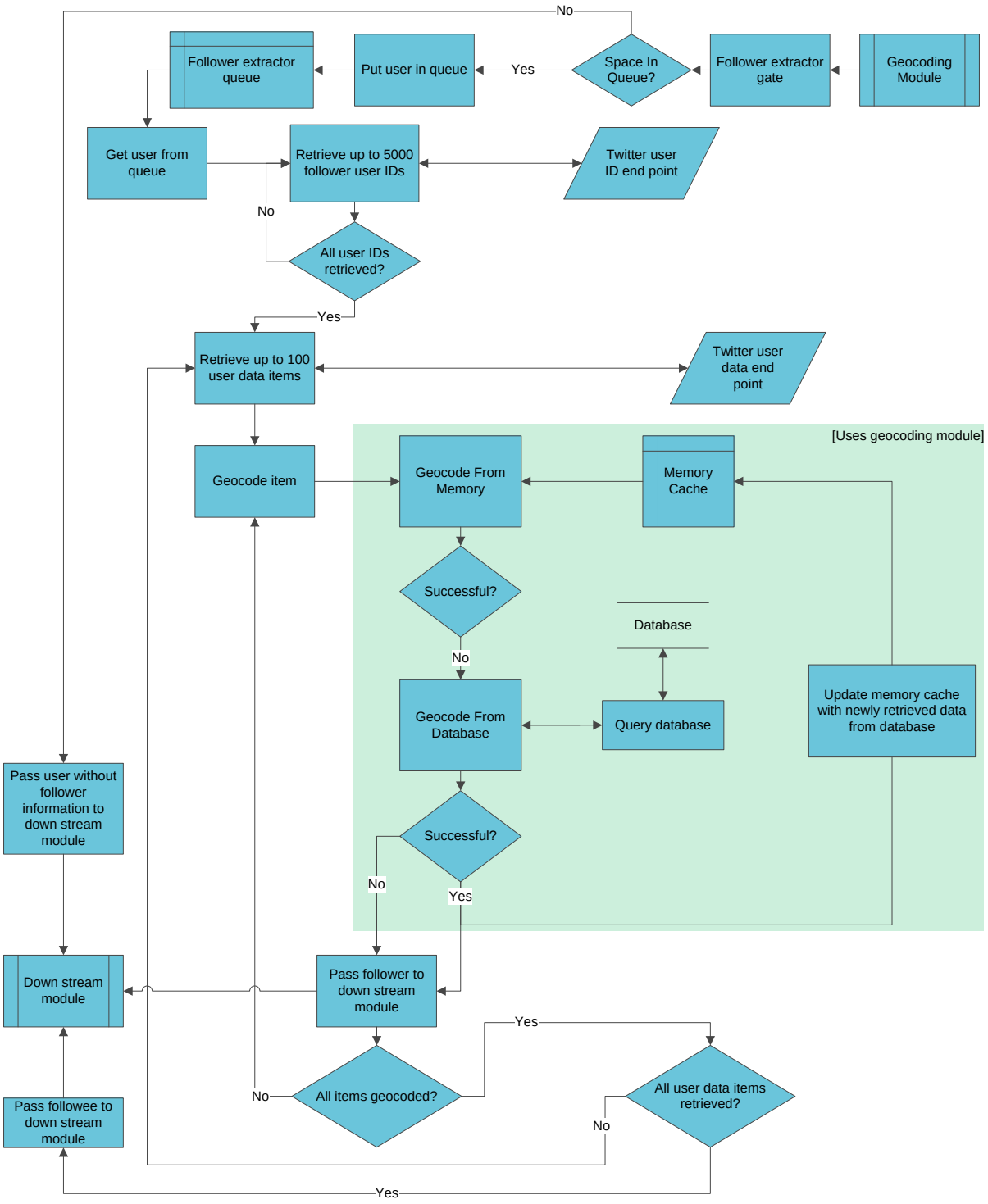


Figure A.10: Flow chart showing the follower enrichment process.

A.11 Figure 11

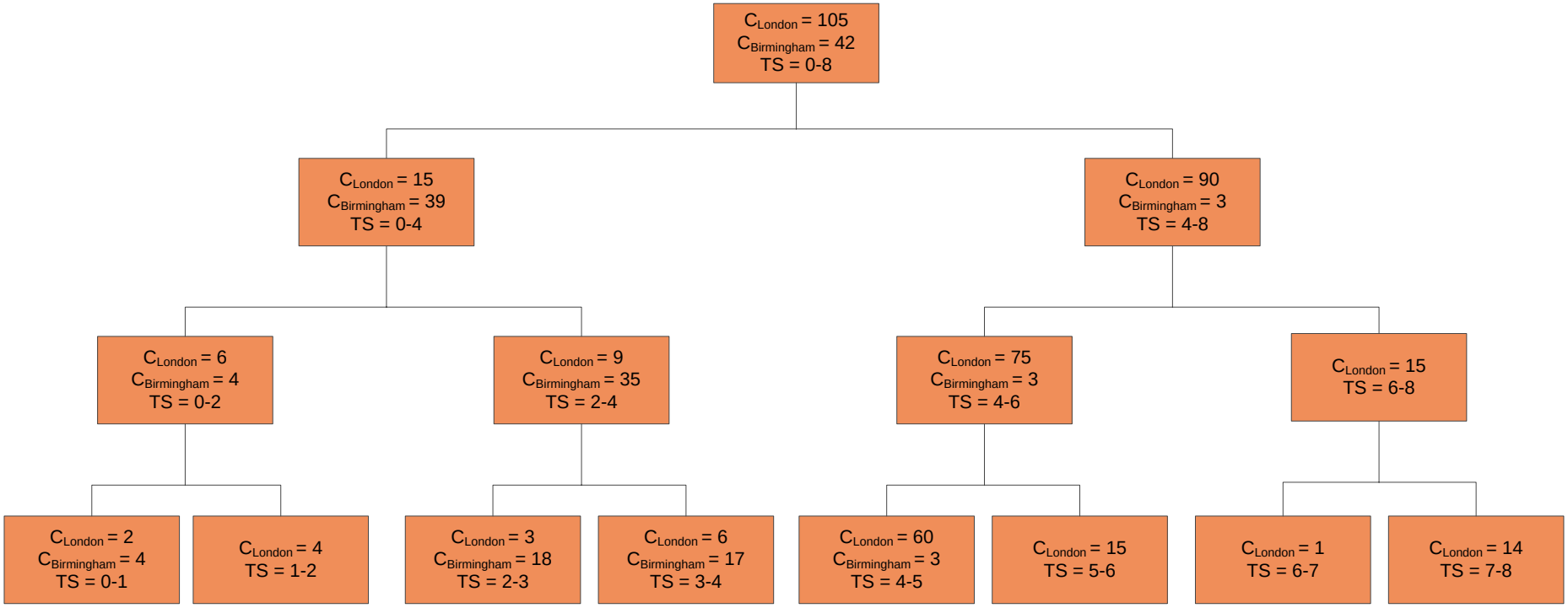


Figure A.11: Tree diagram of the binary tree used to efficiently store spatial degree centrality.

This structure represents the influence of a location on London and Birmingham. TS stands for time step and for example $C_{London} = 6$, $C_{Birmingham} = 4$, $TS = 0 - 2$ indicates that between time step 0 and 2 six followers were geocoded in London and four in Birmingham.

A.12 Figure 12

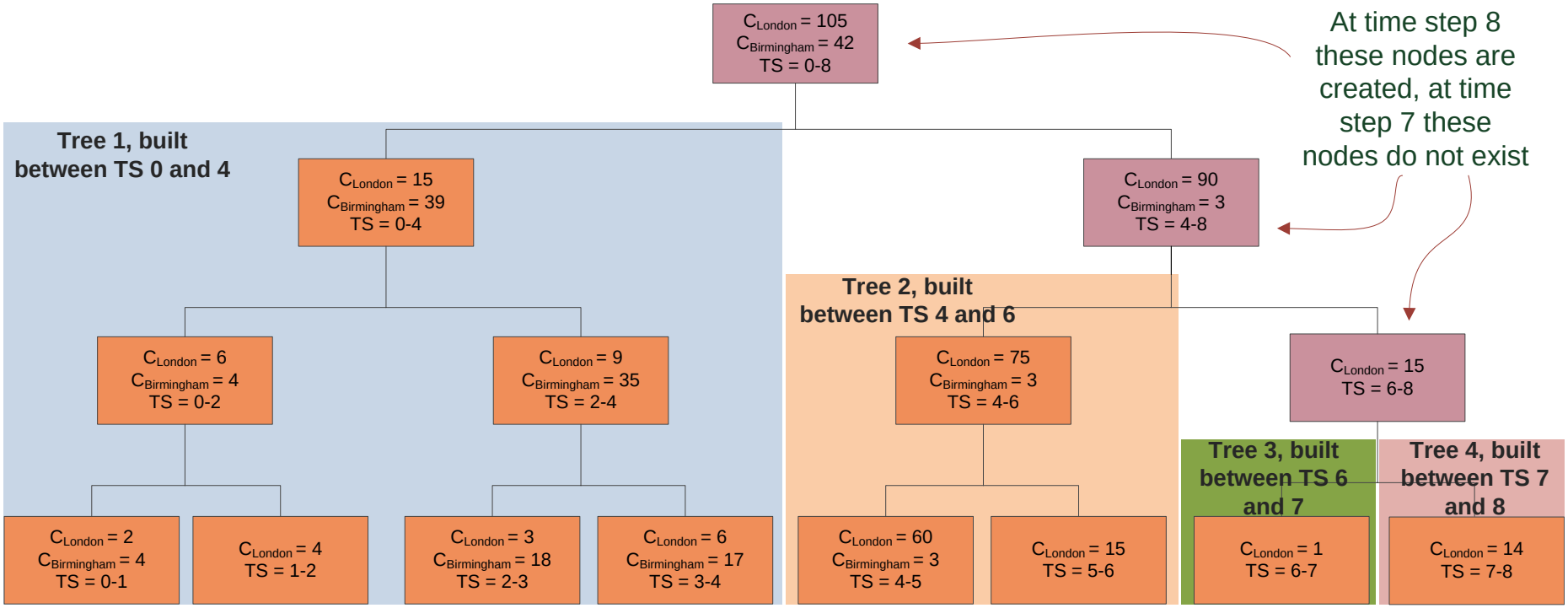


Figure A.12: Tree diagram of the binary tree used to efficiently store spatial degree centrality.

This structure represents the influence of a location on London and Birmingham. TS stands for time step and for example $C_{London} = 6$, $C_{Birmingham} = 4$, $TS = 0 - 2$ indicates that between time step 0 and 2 six followers were geocoded in London and four in Birmingham.

The tree grows from left to right and the tree is initially segregated into multiple separate trees until a new root node is created.

A.13 Figure 13

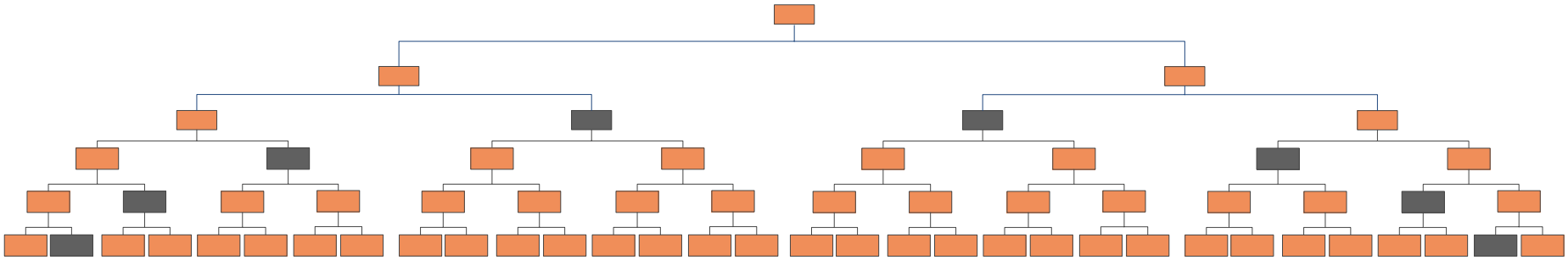


Figure A.13: Tree diagram of the binary tree used to efficiently store spatial degree centrality.

This is a tree spanning 32 time steps with older time steps to the left and newer to the right. Highlighted are nodes which would need to be accessed for a search from time step 1 to 31 which is a worst case search where 8 nodes are accessed. If time step 0 to 32 was searched, only the root node would need to be accessed.

A.14 Figure 14

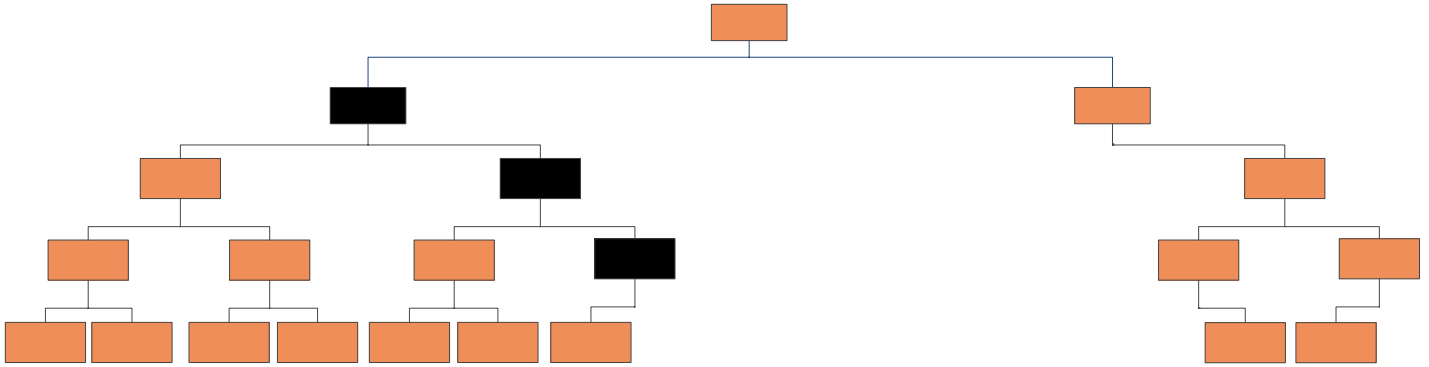


Figure A.14: Tree diagram of the binary tree used to efficiently store spatial degree centrality.

This diagram demonstrates empty nodes being omitted to save storage space. The highlighted black nodes are parents which would normally have been generated upon the completion of an omitted time step (a leaf node is missing).

A.15 Figure 15

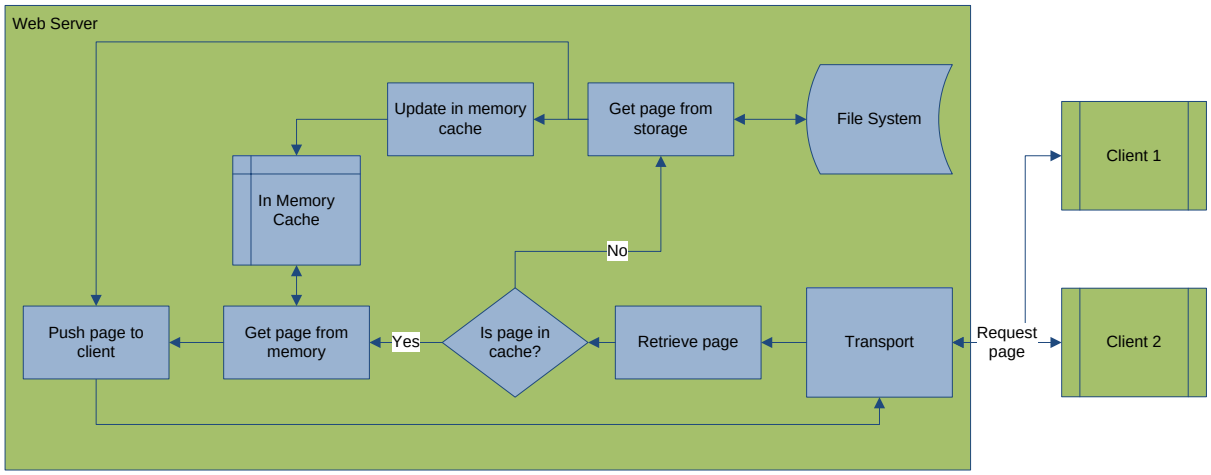


Figure A.15: High level diagram of static page caching.

If both clients are requesting the same page and that page is not already in the cache then the first request will read it from the file system and the second request will read directly from the in memory cache. The second request and all subsequent requests which use the cache will be significantly faster since they do not have the overhead of reading from storage.

A.16 Figure 16

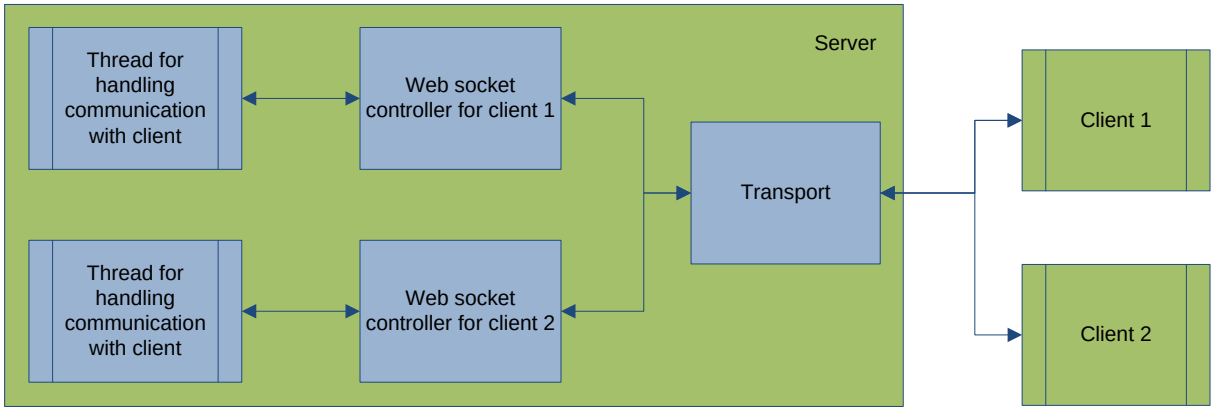


Figure A.16: High level diagram of web sockets.

Clients open up a two way communication channel via a web socket. Each web socket is separate from every other web socket and can remain open indefinitely.

A.17 Figure 17

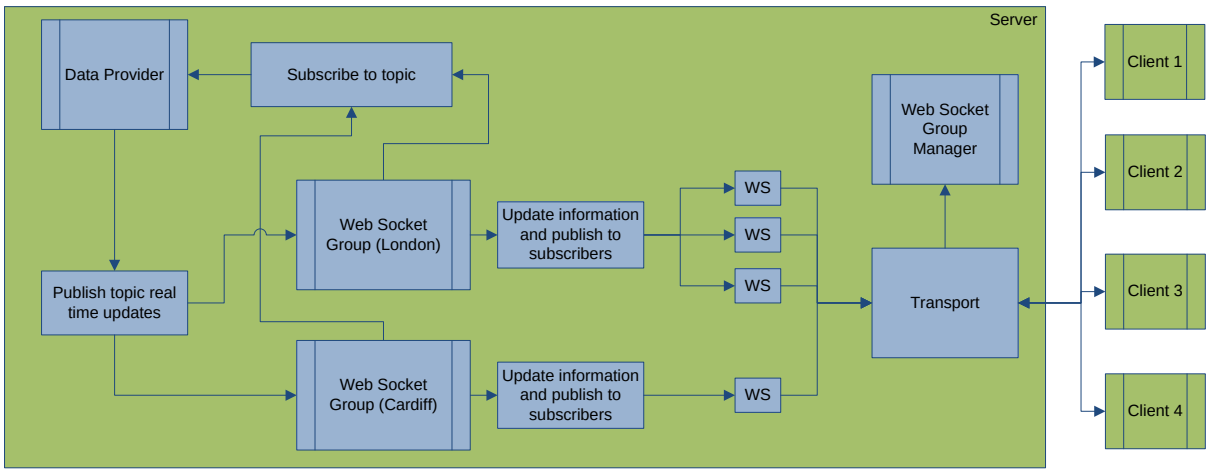


Figure A.17: High level diagram of dynamic page caching.

WS stands for web socket and represents an HTML5 web socket connection. The web socket group manager creates web socket groups as required. In this case there is a web socket group for each city and clients request tweets associated with a city. When a web socket group is created it subscribes to the topic from the data provider. The data provider then notifies its subscribers of changes to topics. Three clients are listening for tweets associated with London; the modifications to the clients views are computed once and the same data is propagated to each client.

A.18 Figure 18

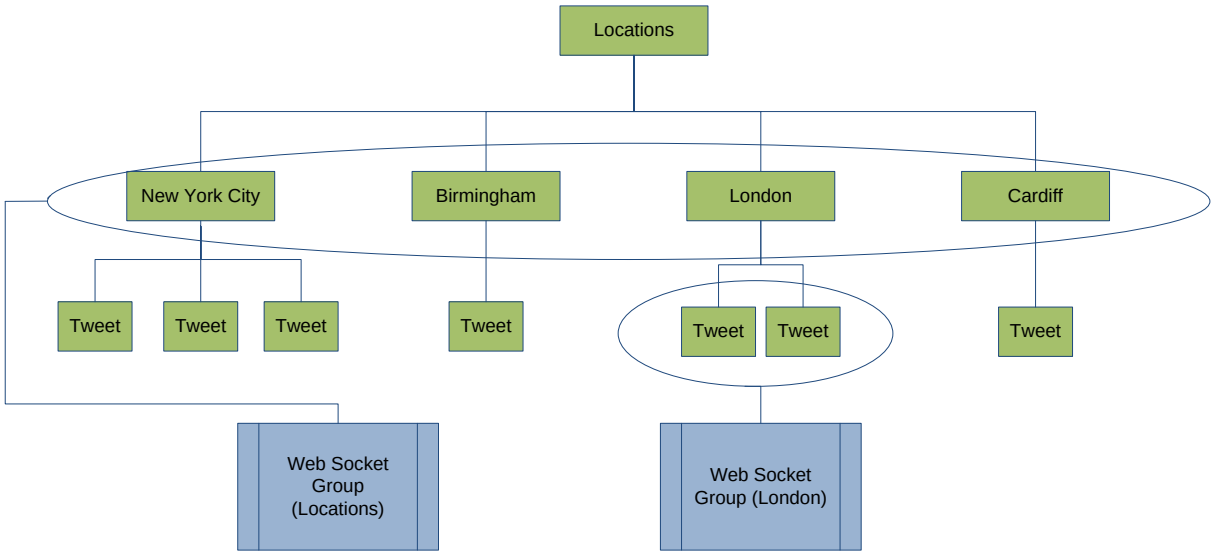


Figure A.18: Shows the data provider tree used by web sockets. Here we have a web socket group which sends a list of locations on the left, it is subscribed to changes in the location branches. On the right we have a web socket group interested in tweets associated with London, it is subscribed to the leaf nodes of the London branch.

A.19 Figure 19

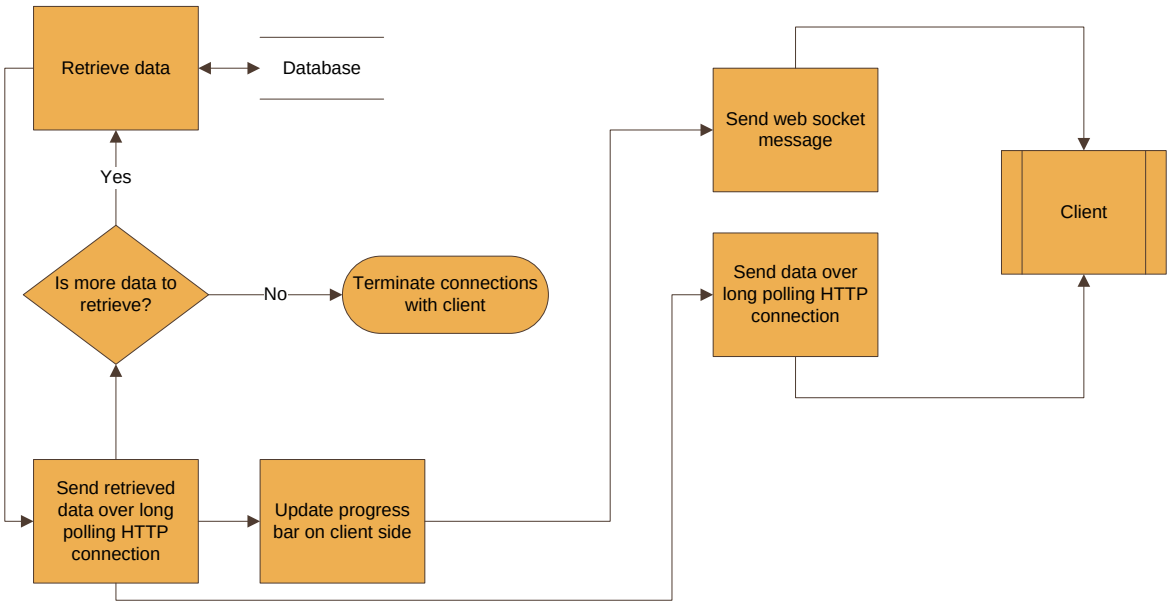


Figure A.19: Flow chart demonstrating the file download "tunnelling" process. A client downloads data from the database and receives records sequentially. Web sockets are used to update progress bars so that the user has an indication as to how long it will take to complete the download.

A.20 Figure 20

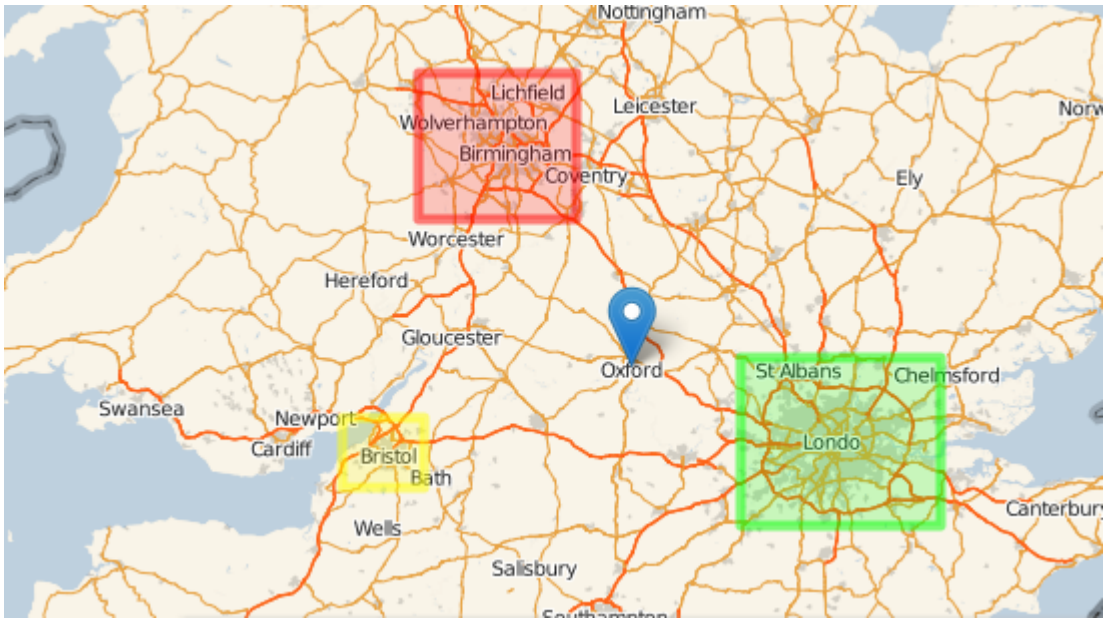


Figure A.20: User drawn geographical criteria for search stream creation.

Tweets geotagged by Twitter as originating from locations within the green box (e.g. London) will be received but their influence won't be analysed.

Users which submit tweets associated with locations within the red box (e.g. Birmingham) will have their influence analysed, but Twitter won't send us all tweets geotagged as originating from within that area; we may receive tweets from users within that area if we are using a keywords filter.

Tweets geotagged by Twitter as originating from locations within the yellow box (e.g. Bristol) will be received and the influence of the users which submitted the tweets will be analysed.

Users which submit tweets associated with Oxford will have their influence analysed; the difference here is we are not selecting a geographical area, instead we are saying precisely that we are using Oxford as an influence source.

A.21 Figure 21

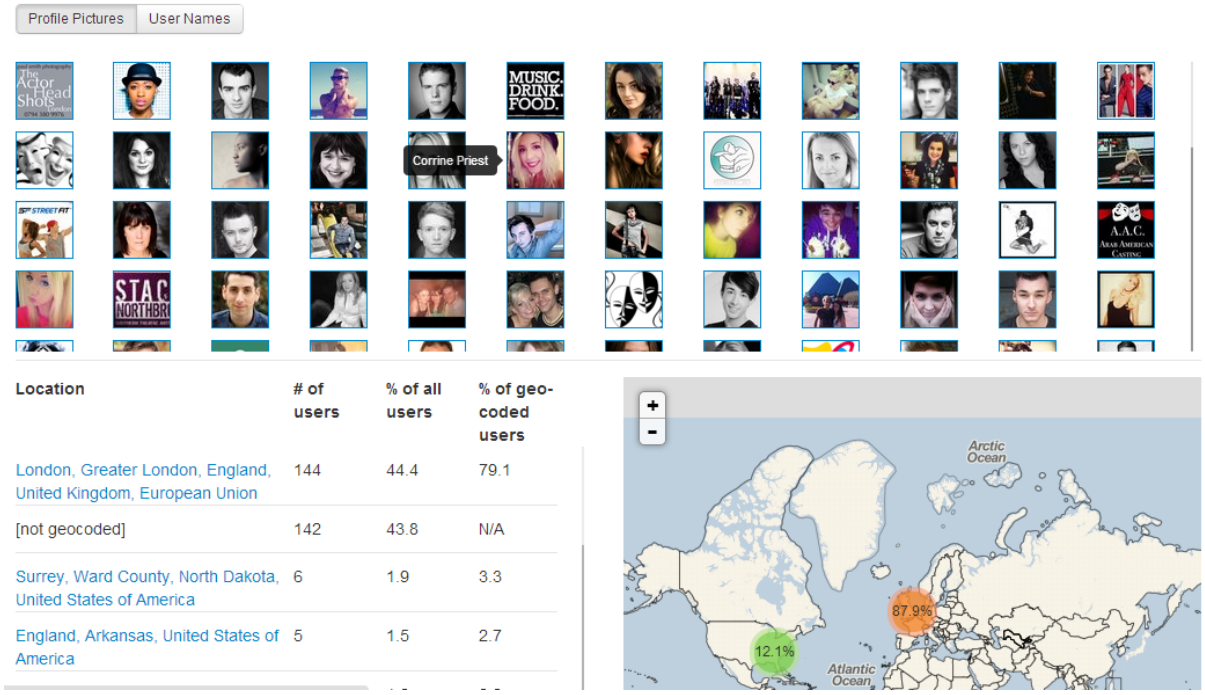


Figure A.21: Screenshot of part of the user page, showing a user's followers in display picture view. The user's influence is also shown as a list and graphically.

A.22 Figure 22

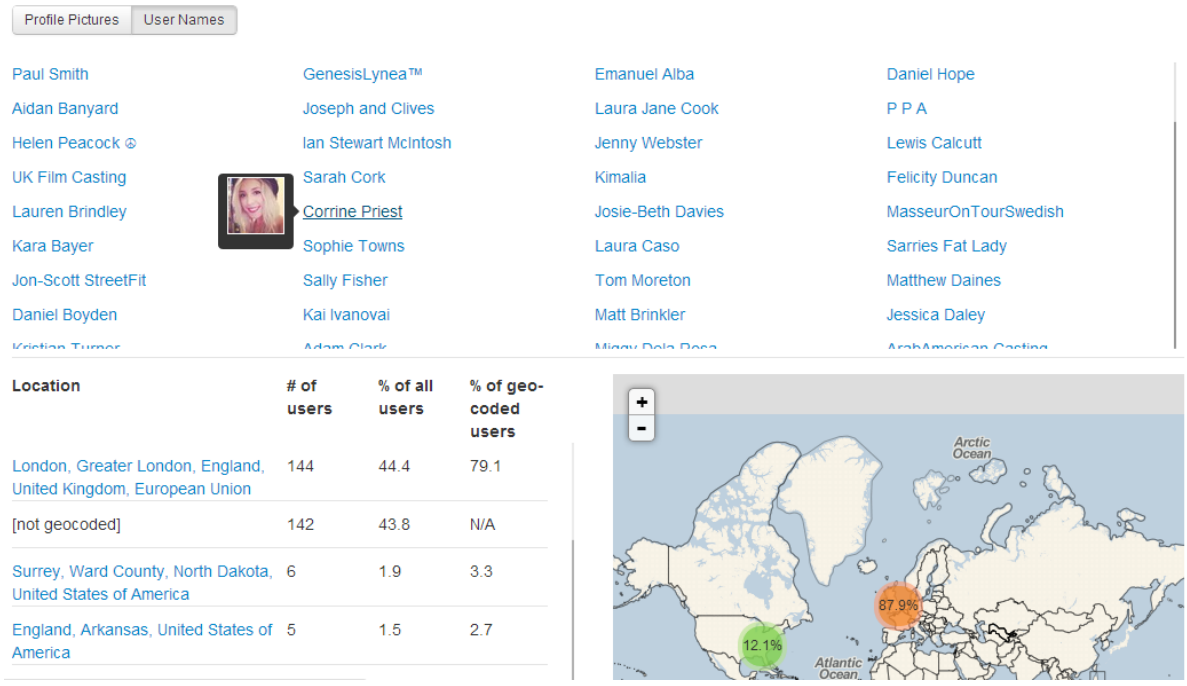


Figure A.22: Screenshot of part of the user page, showing a user's followers in user name view. The user's influence is also shown as a list and graphically.

A.23 Figure 23

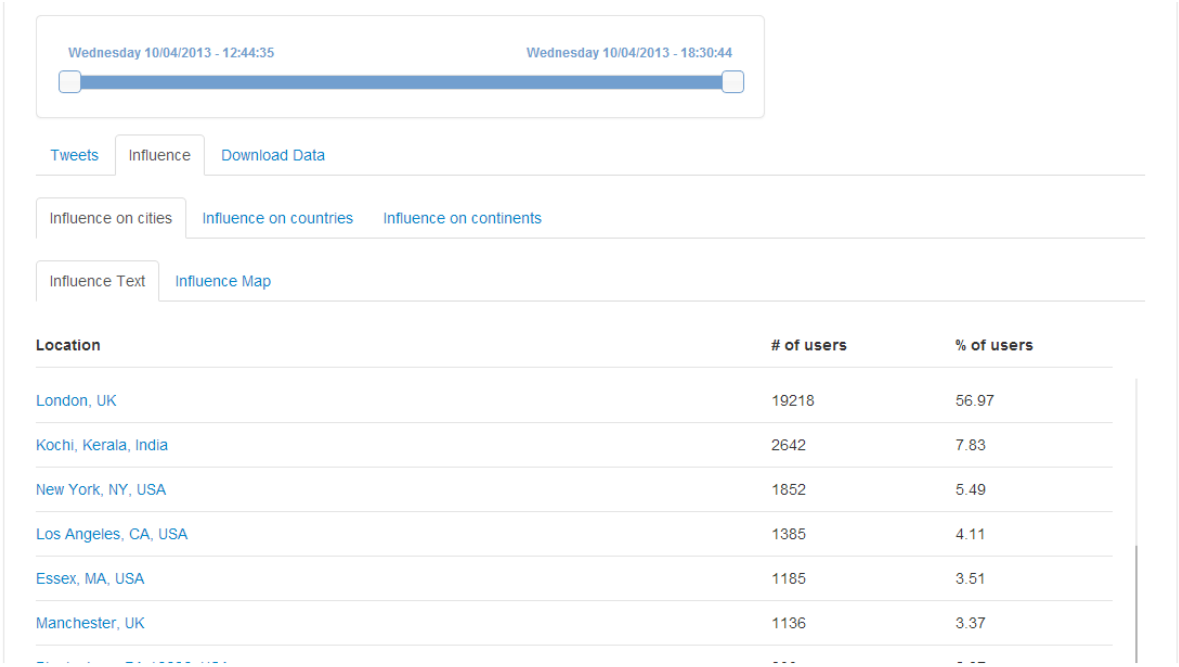


Figure A.23: Screenshot of part of the location page, showing influence of a source location in text form. Slider bar is also shown, moving the slider bar changes influence data by changing the time period for which data should be shown.

A.24 Figure 24

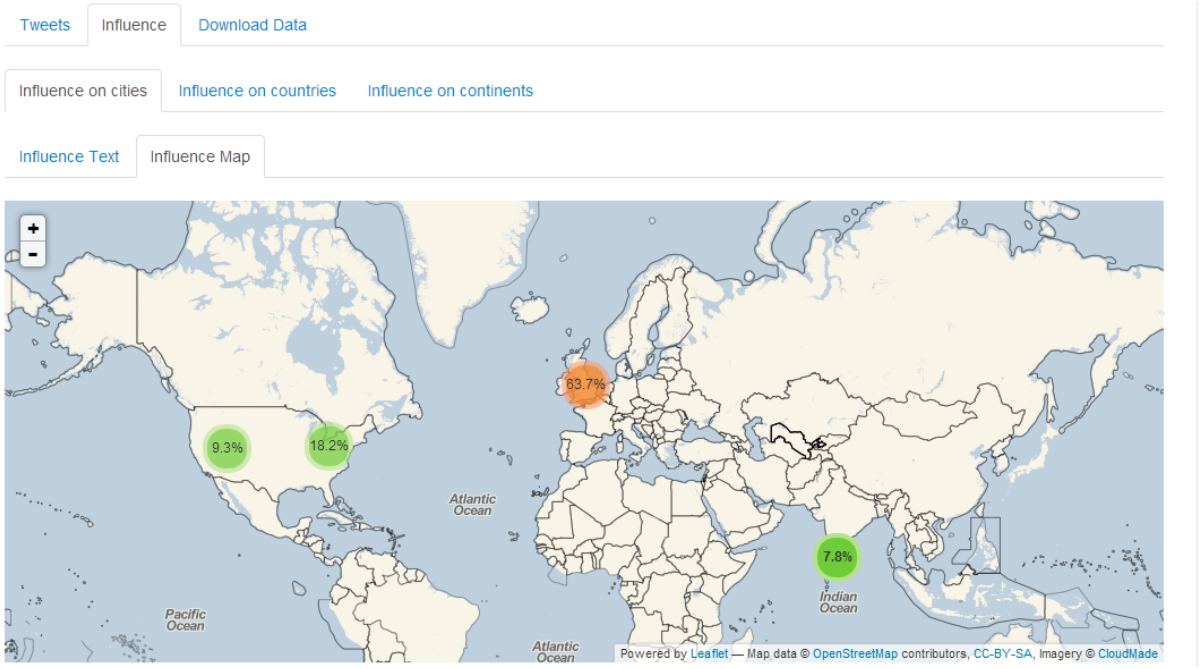


Figure A.24: Screenshot of part of the location page, showing influence of a source location in map form; the percentages correspond to percentage of followers in target location.

A.25 Figure 25

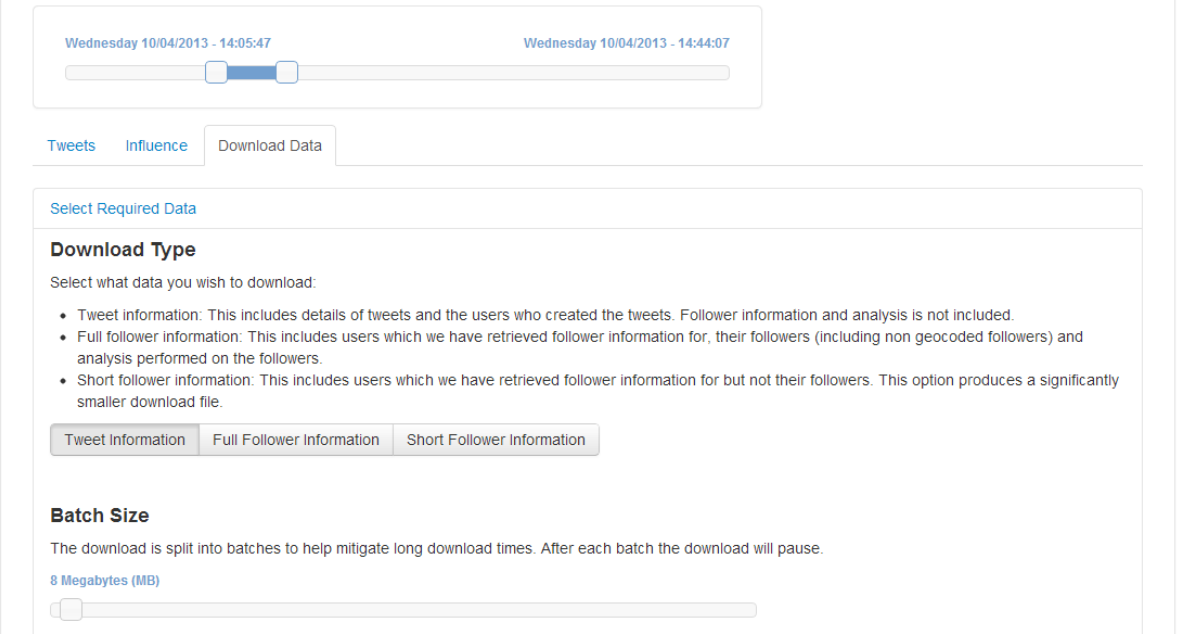


Figure A.25: Screenshot of download data section which is part of location and search stream page. This screen allows clients to download data for specific locations as well as for the entire search stream. This is the setup screen where clients decide what they want to download.

A.26 Figure 26

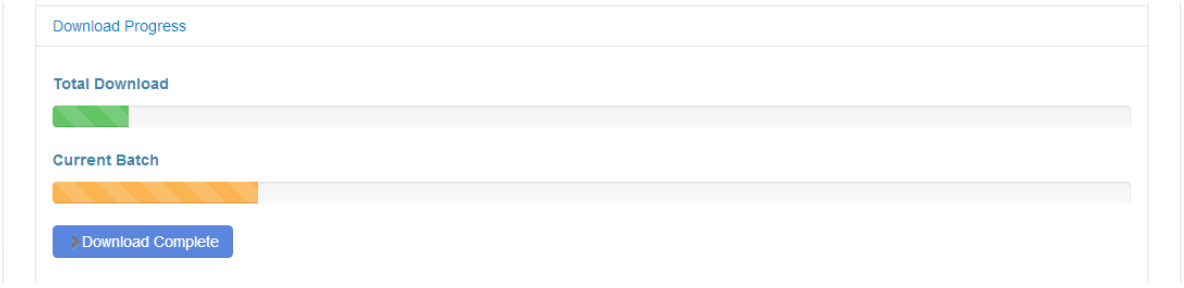


Figure A.26: Screenshot of download data section which is part of location and search stream page. This screen allows clients to download data for specific locations as well as for the entire search stream. This is the progress screen where clients can view how far their download has progressed.

A.27 Figure 27

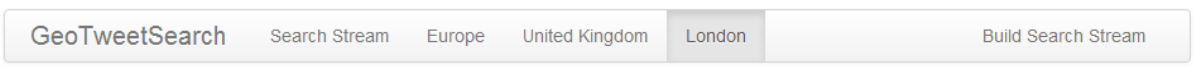


Figure A.27: Screenshot of "breadcrumbs" based navigation. This title bar appears on the top of each page, with different "breadcrumbs" depending on a client's position within the system.

A.28 Figure 28

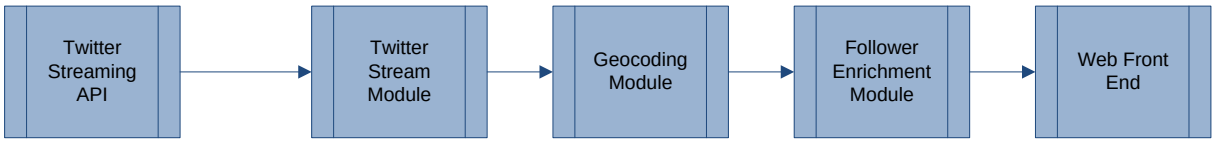


Figure A.28: High level view of how modules discussed in this report fit together.

A.29 Figure 29

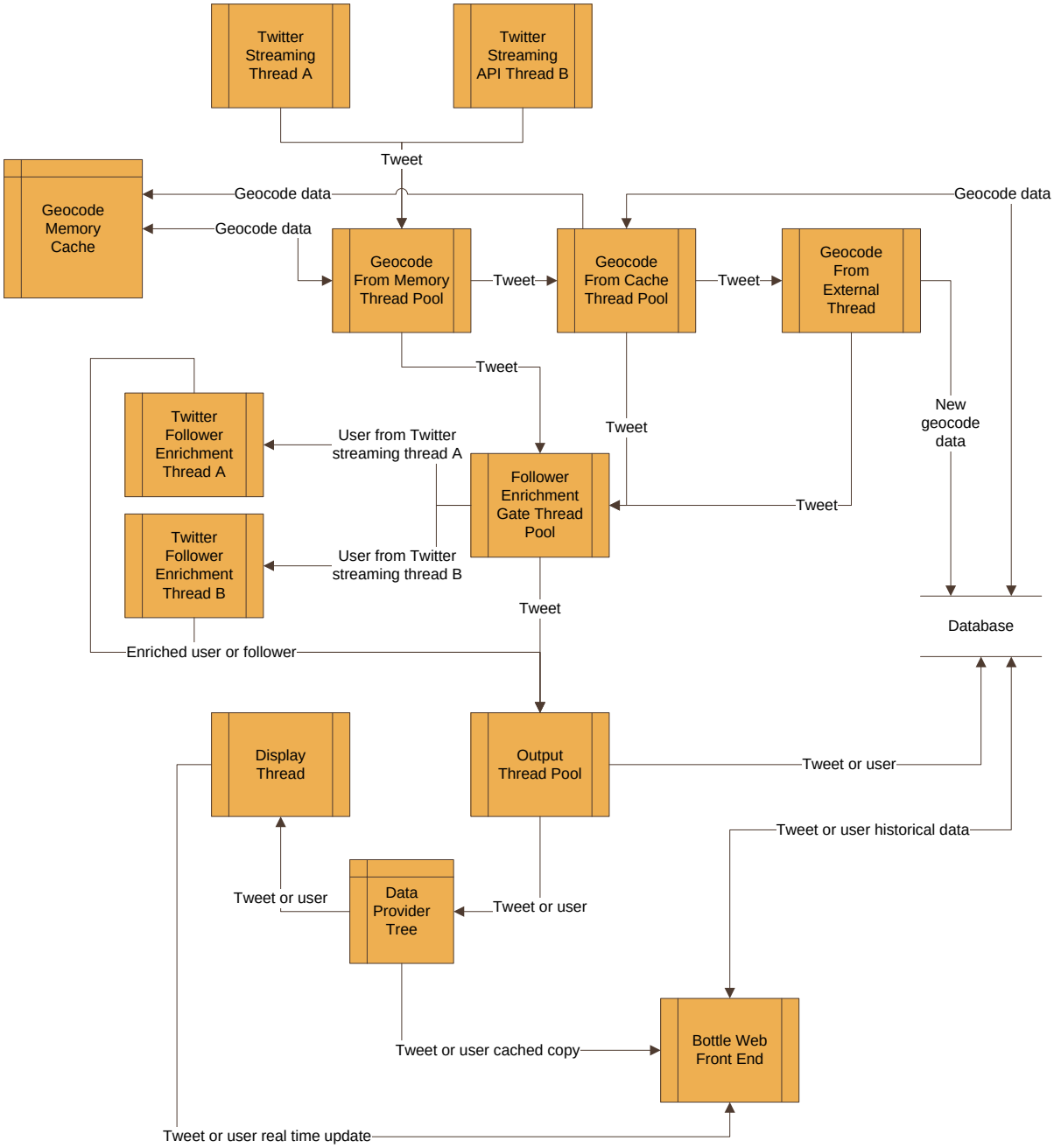


Figure A.29: Flow chart showing how tweet data from the streaming API flows through the system.

A.30 Figure 30

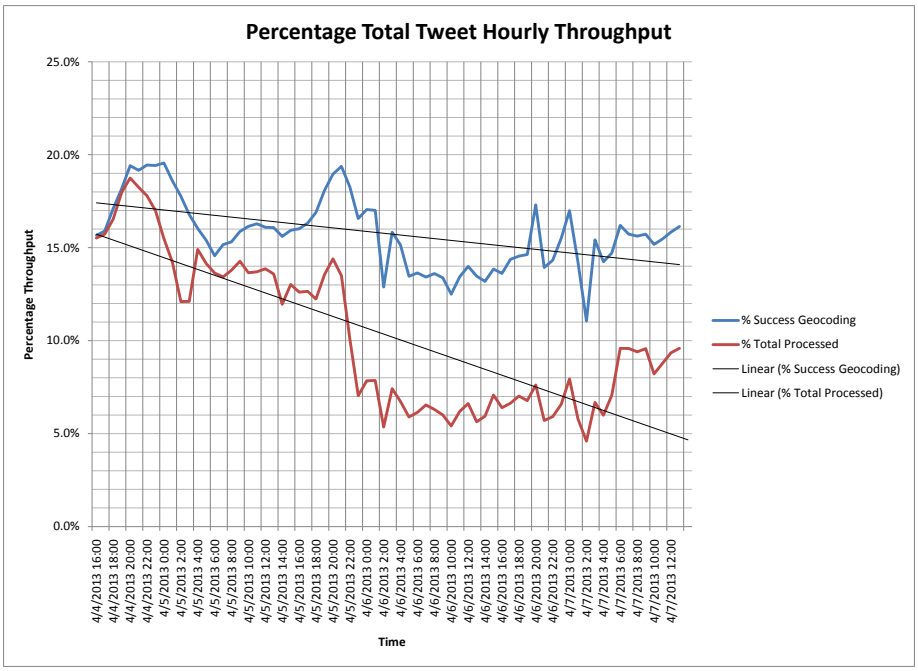


Figure A.30: Graph showing the percentage of tweets from the Twitter streaming API which are fully consumed i.e. geocoded and made available to the user. The graph also shows the percentage of tweets successfully geocoded. The difference between the two lines are tweets being dropped because the system can't handle the load. The graph was built using data gathered over a 70 hour period subscribed to a high traffic stream.

A.31 Figure 31

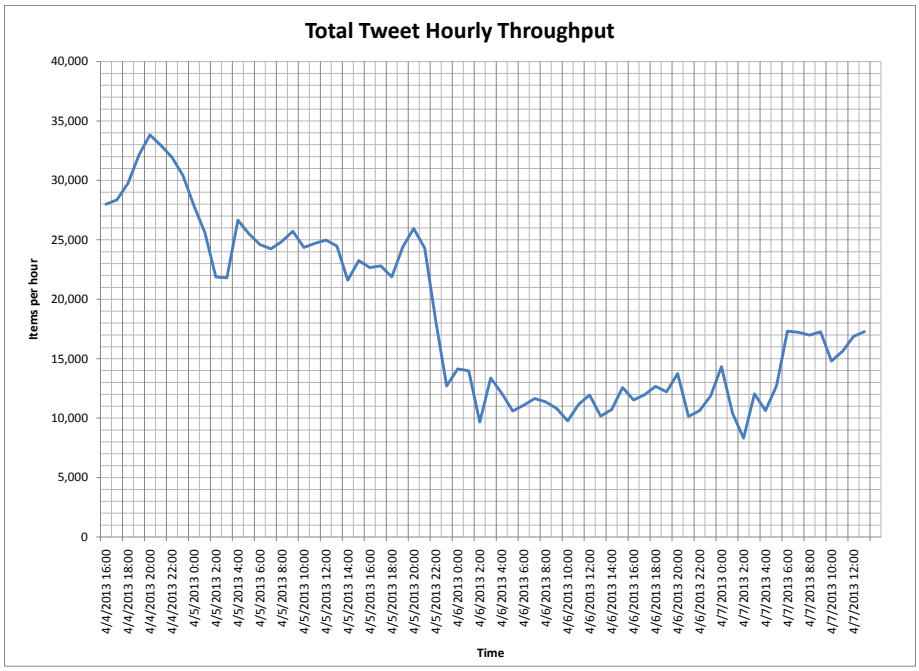
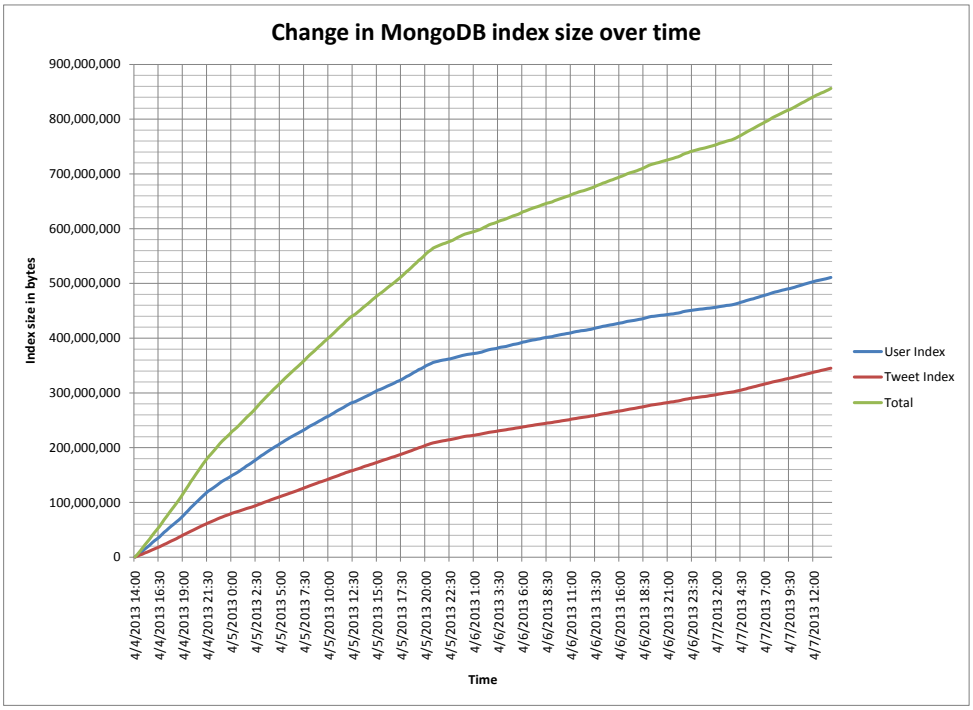


Figure A.31: Graph showing the hourly number of tweets which the system fully processed over a 70 hour period subscribed to a high traffic stream.

A.32 Figure 32



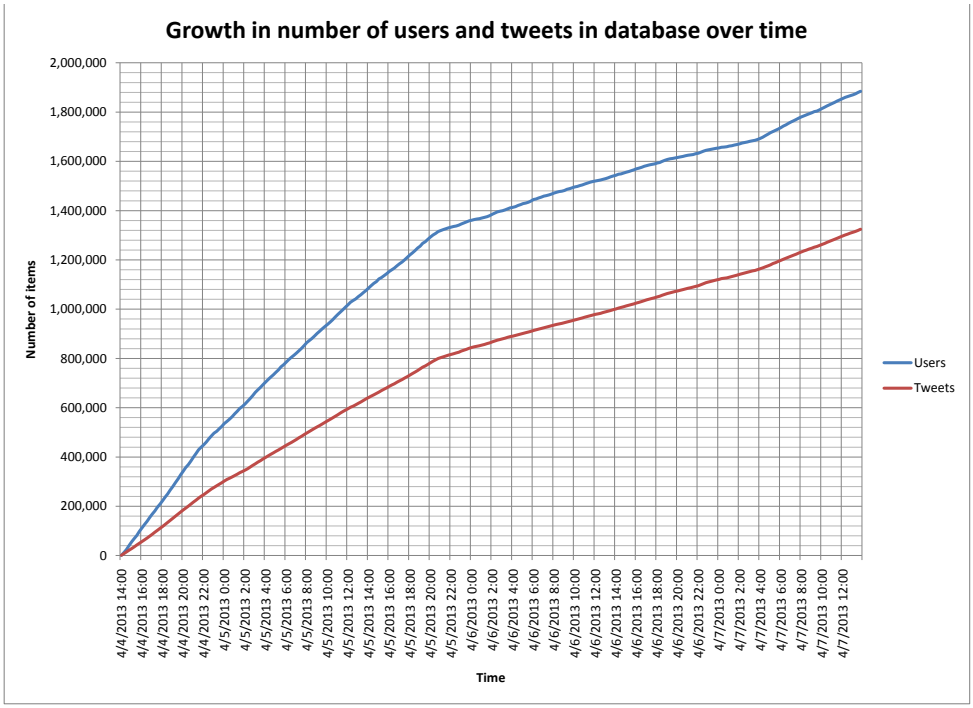


Figure A.32: Graphs showing growth in number of tweets and users stored in the database, as well as growth in the size of their indexes. The graphs were built using data gathered over a 70 hour period subscribed to a high traffic stream.

A.33 Figure 33

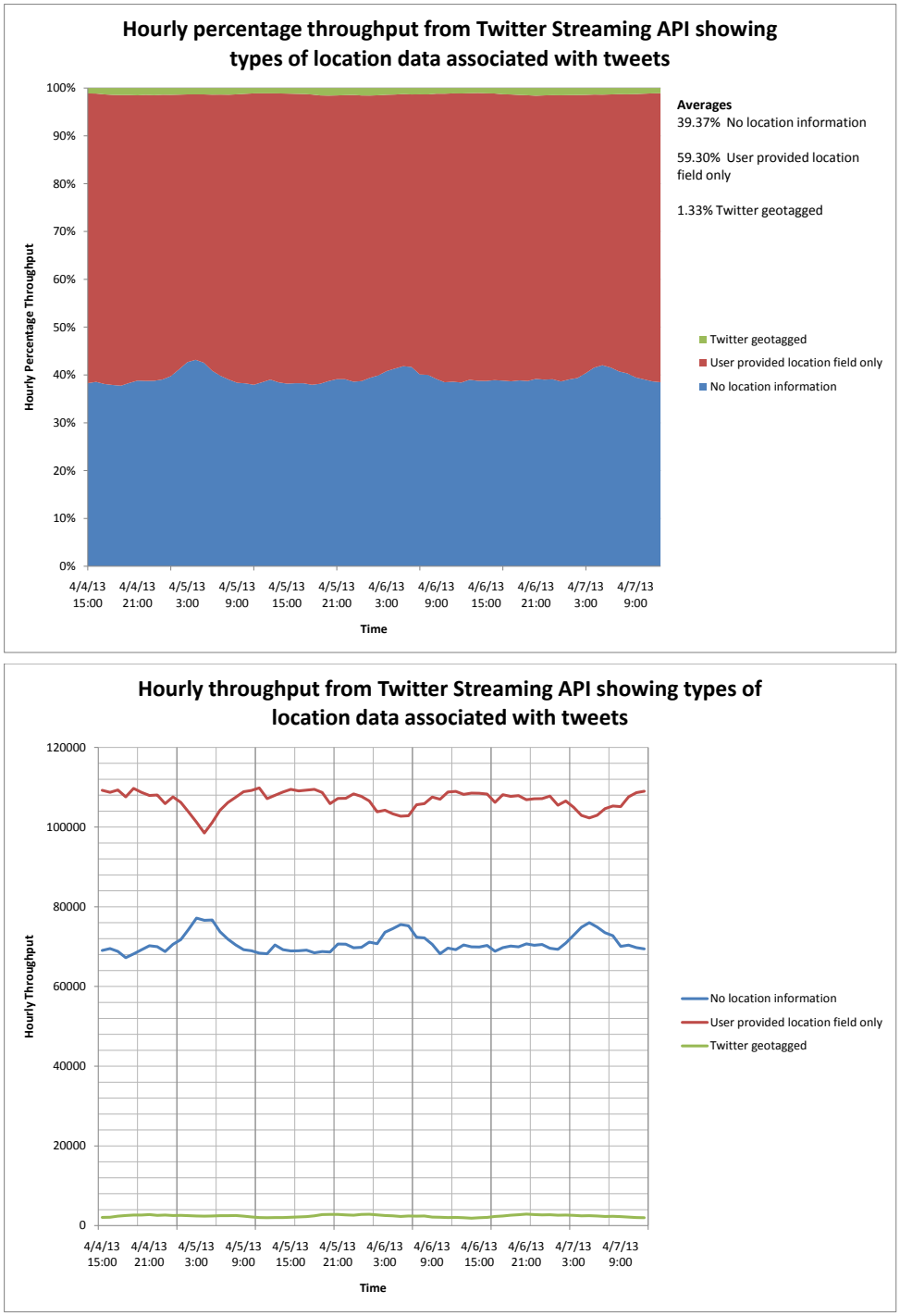


Figure A.33: Graphs showing location data associated with tweets received from the Twitter streaming API. Twitter geotagged tweets include geocoding information provided directly by Twitter. Tweets in the "user provided location field only" category are not geotagged and the user which created the tweet has a user provided location field associated with his account. Tweets in the "no location information" category are not geotagged and the user which created the tweet has no location information associated with their account. All tweets received fall into one of these three categories. The graphs were built using data gathered over a 70 hour period subscribed to a high traffic stream.

Appendix B

Tables

B.1 Table 1

Time period	2013-04-04 14:00 - 2013-04-07 12:00
Received	12,599,337
Rate limited	106,614,590
Total	119,213,927
Percentage received	10.57%
Average hourly received	179,991

Table B.1: Table showing throughput of the Twitter streaming API. Rate limited tweets are tweets that were in the stream but not sent to our application because of the rate limit. Received tweets are tweets which our application actually received through the API. The table was built using data gathered over a 70 hour period subscribed to a high traffic stream.

B.2 Table 2

Twitter user location fields.
London
London, UK
L-O-N-D-O-N
I live in London with my friend
not Cardiff, now live in London
Londres

Table B.2: Table highlighting example user location fields which should all geocode to the same location (London, UK).

B.3 Table 3

	Number	Percentage
Total geocoded from memory	1,980,606	99.84%
Total geocoded from database	1,641	0.08%
Total geocoded from external	1,535	0.08%
Total	1,983,782	

Table B.3: Table showing the percentage of tweets geocoded at each stage of the geocoding process. The table was built using data gathered over a 70 hour period subscribed to a high traffic stream via the Twitter streaming API. Note that the database value is low because when analysis begun we had already loaded a large number of geocode entries into the in memory cache.

B.4 Table 4

Follower enrichment	
Average number of followers per hour	16671.4
Average number of followees per hour	29.7
Average percentage of followers geocoded hourly	28.31%

Table B.4: Table showing number of followees and followers processed by follower enrichment thread. Note that a followee is a user with followers.

B.5 Table 5

M-A-G-I-C	LAND	IS GOOD	Hello World, I live in London sometimes
magic land is good			hello world i live
magic land is			hello world i
magic land			hello world
magic			hello
land is good			world i live in
land is			world i live
land			world i
is good			world
is			i live in london
good			i live in
			i live
			i
			live in london sometimes
			live in london
			live in
			live
			in london sometimes
			in london
			in
			london sometimes
			london

Table B.5: Table showing the geocode queries which take place given two inputs.

B.6 Table 6

Worst Case Complexity			
	No Binary Tree	**Normal Binary Tree**	**Optimized Binary Tree**
Read complexity	$\mathrm{O}(n \cdot \log_2 m)$	$\mathrm{O}(2 \cdot \log_2 m \cdot \log_2 m)$	$\mathrm{O}(x \cdot \log_2 m)$
Write complexity	$\mathrm{O}(\log_2 m)$	$\mathrm{O}(\log_2 m \cdot \log_2 m)$	$\mathrm{O}(\log_2 m \cdot \log_2 m)$
Example read	245760	450	60
Example write	15	225	225

Best Case Complexity			
	No Binary Tree	**Normal Binary Tree**	**Optimized Binary Tree**
Read complexity	$\mathrm{O}(n \cdot \log_2 m)$	$\mathrm{O}(\log_2 m)$	$\mathrm{O}(\log_2 m)$
Write complexity	$\mathrm{O}(\log_2 m)$	$\mathrm{O}(\log_2 m \cdot \log_2 m)$	$\mathrm{O}(\log_2 m)$
Example read	245760	15	15
Example write	15	225	15

Complexity Summary			
	No Binary Tree	**Normal Binary Tree**	**Optimized Binary Tree**
Example read worst — best	245760 — 245760	450 — 15	60 — 15
Example write worst — best	15 — 15	225 — 225	225 — 15

Explanation		
Variable	**Value**	**Meaning**
n	16384	Number of time steps
m	32768	Number of nodes in structure
x	4	Maximum number of queries
$log_2(m)$	15	Height of binary tree
		Database read/write node in binary tree.

Table B.6: A set of tables comparing time complexities (big O notation) of different methods of reading, writing and storing temporal influence data in a database. An example is provided with the variables shown in the explanation table; the example results indicate the expected number of operations required to read or write data.

Note that it is assumed that all nodes have either 0 or 2 children i.e. it is a full binary tree. It is assumed that the tree is stored in a database with efficient indexing i.e. logarithmic read and write complexity. In addition it is assumed that this tree is the only tree in the index and as a result requires $\mathrm{O}(\log_2 m)$ complexity to lookup, update or create a single node in the tree.

B.7 Table 7

Python Packages			
Package	**Version**	**Install Via**	**Website**
gevent	1.0 RC3	gevent website	http://www.gevent.org/
gevent-websocket	0.3.6	pip	http://www.gelens.org/code/gevent-websocket/
greenlet	0.4.0	pip	http://greenlet.readthedocs.org/en/latest/
requests	1.1.0	pip	http://docs.python-requests.org/en/latest/
oauthlib	0.3.8	pip	https://readthedocs.org/projects/oauthlib/
requests-oauthlib	0.3.0	pip	https://github.com/requests/requests-oauthlib
pymongo	2.4.2	pip	http://api.mongodb.org/python/current/
bottle	0.11.6	pip	http://bottlepy.org/docs/stable/

JavaScript Plugins		
Plugin	**Respository Path**	**Website**
jQuery	jquery	http://jquery.com/
bootstrap	bootstrap	http://twitter.github.io/bootstrap/
Leaflet	leaflet	http://leafletjs.com/
Leaflet.markercluster	leaflet	https://github.com/Leaflet/Leaflet.markercluster
jQuery UI	jquery-ui	http://jqueryui.com/
jQuery Cookie	jquery-cookie	https://github.com/carhartl/jquery-cookie
jQuery Custom Content Scroller	jquery-custom-scrollbar	http://manos.malihu.gr/jquery-custom-content-scroller/
jQuery Mouse Wheel	jquery-mousewheel	https://github.com/brandonaaron/jquery-mousewheel

Table B.7: Table showing Python packages and JavaScript plugins used by this project.

Third party JavaScript plugins are stored with root path "fyp/api/web/static/third-party" in this project's repository.

Websites and installation paths were verified on 10th of April 2013.

Note: Pip is a Python repository.

Appendix C

Miscellaneous

C.1 Running Project

To run the project with default arguments simply run:

```
python main.py
```

To run the project with a different configuration file use the config switch:

```
python main.py --config my_config.conf
```

To run the project with a different logging configuration file use the logging config switch:

```
python main.py --logging_config my_logging_config.conf
```

Search streams are preserved even after killing the process, to start cleanly run:

```
python main.py --wipe_instance_data
```

Geocode data is also preserved, to wipe geocode data during startup run:

```
python main.py --clear_geocode_data
```

A number of other arguments exist but are intended to aid development only and should otherwise be ignored. For a full list of these arguments and a brief explanation of their purpose run:

```
python main.py --help
```

When the project is running, use your browser to load the URL: `http://127.0.0.1:8000`[1]. The project was tested using the latest version of Google Chrome and Mozilla Firefox.

First time users should note that packages listed in table B.7 are required. In addition a MongoDB instance is required; please modify the configuration file as necessary to point to your MongoDB instance.

[1] This assumes that you are using the default configuration file.

Bibliography

Anttonen, M., Salminen, A., Mikkonen, T., and Taivalsaari, A. (2011). Transforming the web into a real application platform: new technologies, emerging trends and missing pieces. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 800–807. ACM.

Apache (2013). In-memory caching. *http://httpd.apache.org/docs/2.2/caching.html#inmemory*. [Online; accessed: 9 April 2013].

Beevolve (2012). An exhaustive study of twitter users across the world. *http://www.beevolve.com/twitter-statistics/*. [Online; accessed: 4 April 2013].

Bootstrap (2013). Twitter bootstrap. *http://twitter.github.io/bootstrap/index.html*. [Online; accessed: 10 April 2013].

Bottle (2013). Bottle: Python web framework. *http://bottlepy.org/docs/stable/*. [Online; accessed: 10 April 2013].

Bozdag, E., Mesbah, A., and Van Deursen, A. (2009). Performance testing of data delivery techniques for ajax applications. *Journal of Web Engineering*, 8(4):287–315.

Brown, C. M. (1998). *Human-computer interface design guidelines*. Intellect Books.

Cao, X., Cong, G., and Jensen, C. S. (2010). Mining significant semantic locations from gps data. *Proc. VLDB Endow.*, 3(1-2):1009–1020.

Chodorow, K. and Dirolf, M. (2010). *MongoDB: The Definitive Guide*. O'Reilly Media.

Chun, W. (2001). *Core Python Programming: Text*. Prentice Hall Ptr Core Series. Prentice Hall Ptr.

CloudMade (2013). Map tiles. *http://developers.cloudmade.com/projects/show/tiles*. [Online; accessed: 10 April 2013].

Collinson, M. (2009). Geographic names database country bounds. *http://wiki.openstreetmap.org/wiki/User:Ewmjc/Country_bounds*. [Online; accessed: 6 April 2013].

Comer, D. (1979). Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137.

Drummond, W. J. (1995). Address matching: Gis technology for mapping human activity patterns. *Journal of the American Planning Association*, 61(2):240–251.

Eng, J. (2003). Sample size estimation: How many individuals should be studied? 1. *Radiology*, 227(2):309–313.

Facebook (2013a). API login. *https://developers.facebook.com/docs/reference/api/data-access/*. [Online; accessed: 4 April 2013].

Facebook (2013b). API search. *https://developers.facebook.com/docs/reference/api/search/*. [Online; accessed: 4 April 2013].

Farrell, J. and Nezlek, G. S. (2007). Rich internet applications the next stage of application development. In *Information Technology Interfaces, 2007. ITI 2007. 29th International Conference on*, pages 413–418. IEEE.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext transfer protocol – http/1.1.

Foursquare (2013). About foursquare. *https://foursquare.com/about/*. [Online; accessed: 14 April 2013].

Fowler, G. A. (2012). Facebook: One billion and counting. *The Wall Street Journal*. [Online; accessed: 4 April 2013].

gevent (2013). What is gevent? *http://www.gevent.org/*. [Online; accessed: 10 April 2013].

Gift, N. (2008). Practical threaded programming with python. *"http://www.ibm.com/developerworks/aix/library/au-threadingpython/"*. [Online; accessed: 11 April 2013].

Google (2013a). The google geocoding api. *https://developers.google.com/maps/documentation/geocoding*. [Online; accessed: 5 April 2013].

Google (2013b). API activities. *https://developers.google.com/+/api/latest/activities*. [Online; accessed: 4 April 2013].

Google (2013c). API login. *https://developers.google.com/+/api/oauth*. [Online; accessed: 4 April 2013].

Greenlet (2013). greenlet: Lightweight concurrent programming. *http://greenlet.readthedocs.org/en/latest/*. [Online; accessed: 10 April 2013].

Gube, J. (2009). Breadcrumbs in web design: Examples and best practices. *http://www.smashingmagazine.com/2009/03/17/breadcrumbs-in-web-design-examples-and-best-practices-2/*. [Online; accessed: 14 April 2013].

Hecht, B., Hong, L., Suh, B., and Chi, E. H. (2011). Tweets from justin bieber's heart: the dynamics of the location field in user profiles. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 237–246, New York, NY, USA. ACM.

Heidemann, J., Klier, M., and Probst, F. (2012). Online social networks: A survey of a global phenomenon. *Computer Networks*, 56(18):3866 – 3878. The WEB we live in.

Iyengar, A. and Challenger, J. (1997). *Improving web server performance by caching dynamic data*. IBM TJ Watson Research Center.

Jelenkovi, P. R. and Radovanovi, A. (2004). Least-recently-used caching with dependent requests. *Theoretical Computer Science*, 326(13):293 – 327.

Kaiser, J. and Mock, M. (1999). Implementing the real-time publisher/subscriber model on the controller area network (can). In *Object-Oriented Real-Time Distributed Computing, 1999. (ISORC '99) Proceedings. 2nd IEEE International Symposium on*, pages 172–181.

Kaplan, A. M. and Haenlein, M. (2010). Users of the world, unite! the challenges and opportunities of social media. *Business horizons*, 53(1):59–68.

Kennedy, M. (2010). Mongodb vs. sql server 2008 performance showdown. *http://blog.michaelckennedy.net/2010/04/29/mongodb-vs-sql-server-2008-performance-showdown/*. [Online; accessed: 10 April 2013].

Lawson, B. and Sharp, R. (2011). *Introducing HTML5*. Pearson Education.

Leaflet (2013). An open-source javascript library for mobile-friendly interactive maps. *http://leafletjs.com/*. [Online; accessed: 10 April 2013].

Li, Y.-M. and Shiu, Y.-L. (2012). A diffusion mechanism for social advertising over microblogs. *Decision Support Systems*, 54(1):9 – 22.

Lima, A. and Musolesi, M. (2012). Spatial dissemination metrics for location-based social networks. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 972–979. ACM.

Markatos, E. P. (1996). Main memory caching of web documents. *Computer Networks and ISDN Systems*, 28(711):893 – 905. Proceedings of the Fifth International World Wide Web Conference 6-10 May 1996.

MarkerCluster (2013). Leaflet.markercluster. *https://github.com/Leaflet/Leaflet.markercluster*. [Online; accessed: 10 April 2013].

Mikkonen, T. and Taivalsaari, A. (2008). Towards a uniform web application platform for desktop computers and mobile devices. Technical report, Mountain View, CA, USA.

MongoDB (2013a). Indexing overview. *http://docs.mongodb.org/manual/core/indexes/*. [Online; accessed: 10 April 2013].

MongoDB (2013b). Sharded cluster overview. *http://docs.mongodb.org/manual/core/sharded-clusters/*. [Online; accessed: 11 April 2013].

MongoDB (2013c). What happens if an index does not fit into ram? *http://docs.mongodb.org/manual/faq/indexes/#what-happens-if-an-index-does-not-fit-into-ram*. [Online; accessed: 11 April 2013].

OAuthLib (2013). Requests-oauthlib. *https://github.com/requests/requests-oauthlib*. [Online; accessed: 10 April 2013].

OpenMapQuest (2013). Openmapquest nominatim search service developer's guide. *http://open.mapquestapi.com/nominatim/*. [Online; accessed: 6 April 2013].

PyMongo (2013). Pymongo documentation. *https://github.com/requests/requests-oauthlib*. [Online; accessed: 10 April 2013].

Rainie, H., Rainie, L., and Wellman, B. (2012). *Networked: The new social operating system*. MIT Press.

Rajkumar, R., Gagliardi, M., and Sha, L. (1995). The real-time publisher/subscriber interprocess communication model for distributed real-time systems: design and implementation. In *Real-Time Technology and Applications Symposium, 1995. Proceedings*, pages 66–75. IEEE.

Reitz, K. (2013). Requests: Http for humans. *http://www.python-requests.org/en/latest/*. [Online; accessed: 10 April 2013].

Roongpiboonsopit, D. and Karimi, H. A. (2010). Comparative evaluation and analysis of online geocoding services. *International Journal of Geographical Information Science*, 24(7):1081–1100.

Scellato, S., Mascolo, C., Musolesi, M., and Crowcroft, J. (2011). Track globally, deliver locally: improving content delivery networks by tracking geographic social cascades. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 457–466, New York, NY, USA. ACM.

Shaffer, C. A. (2001). *A Practical Introduction to Data Structures and Algorithm Analysis.* Prentice Hall.

Shih, B.-Y., Chen, C.-Y., and Chen, Z.-S. (2012). An empirical study of an internet marketing strategy for search engine optimization. *Human Factors and Ergonomics in Manufacturing & Service Industries.*

Stutzman, F., Gross, R., and Acquisti, A. (2013). Silent listeners: The evolution of privacy and disclosure on facebook. *Journal of Privacy and Confidentiality*, 4(2):2.

Takhteyev, Y., Gruzd, A., and Wellman, B. (2012). Geography of twitter networks. *Social Networks*, 34(1):73 – 81. Capturing Context: Integrating Spatial and Social Network Analyses.

Twitter (2012). Twitter turns six. *http://blog.twitter.com/2012/03/twitter-turns-six.html*. [Online; accessed: 4 April 2013].

Twitter (2013a). 3-legged authorization. *https://dev.twitter.com/docs/auth/3-legged-authorization*. [Online; accessed: 10 April 2013].

Twitter (2013b). API connecting to a streaming end point. *https://dev.twitter.com/docs/streaming-apis/connecting*. [Online; accessed: 4 April 2013].

Twitter (2013c). API geo developer guidelines. *https://dev.twitter.com/terms/geo-developer-guidelines*. [Online; accessed: 4 April 2013].

Twitter (2013d). API place object. *https://dev.twitter.com/docs/platform-objects/places*. [Online; accessed: 4 April 2013].

Twitter (2013e). API POST statuses/filter. *https://dev.twitter.com/docs/api/1.1/post/statuses/filter*. [Online; accessed: 4 April 2013].

Twitter (2013f). API user object. *https://dev.twitter.com/docs/platform-objects/users*. [Online; accessed: 4 April 2013].

Twitter (2013g). GET followers/ids. *https://dev.twitter.com/docs/api/1.1/get/followers/ids*. [Online; accessed: 7 April 2013].

Twitter (2013h). GET users/lookup. *https://dev.twitter.com/docs/api/1.1/get/users/lookup*. [Online; accessed: 7 April 2013].

Twitter (2013i). Vine: A new way to share video. *http://blog.twitter.com/2013/01/vine-new-way-to-share-video.html*. [Online; accessed: 14 April 2013].

United States Board on Geographic Names (2013). Geographic names database. *http://earth-info.nga.mil/gns/html/namefiles.htm*. [Online; accessed: 6 April 2013].

United States Census Bureau (2013). Topologically integrated geographic encoding and referencing. *http://www.census.gov/geo/maps-data/data/tiger.html*. [Online; accessed: 5 April 2013].

w3schools (2013a). Ajax introduction. *http://www.w3schools.com/jquery/jquery_intro.asp*. [Online; accessed: 10 April 2013].

w3schools (2013b). Ajax introduction. *http://www.w3schools.com/ajax/ajax_intro.asp*. [Online; accessed: 10 April 2013].

w3schools (2013c). Css introduction. *http://www.w3schools.com/css/css_intro.asp*. [Online; accessed: 13 April 2013].

Yamada, A., Kim, T. H.-J., and Perrig, A. (2012). Exploiting privacy policy conflicts in online social networks. Technical report, Technical report, Carnegie Mellon University.

Zuckerberg, M. (2008). Facebook blog post. *https://blog.facebook.com/blog.php?post=28111272130*. [Online; accessed: 4 April 2013].