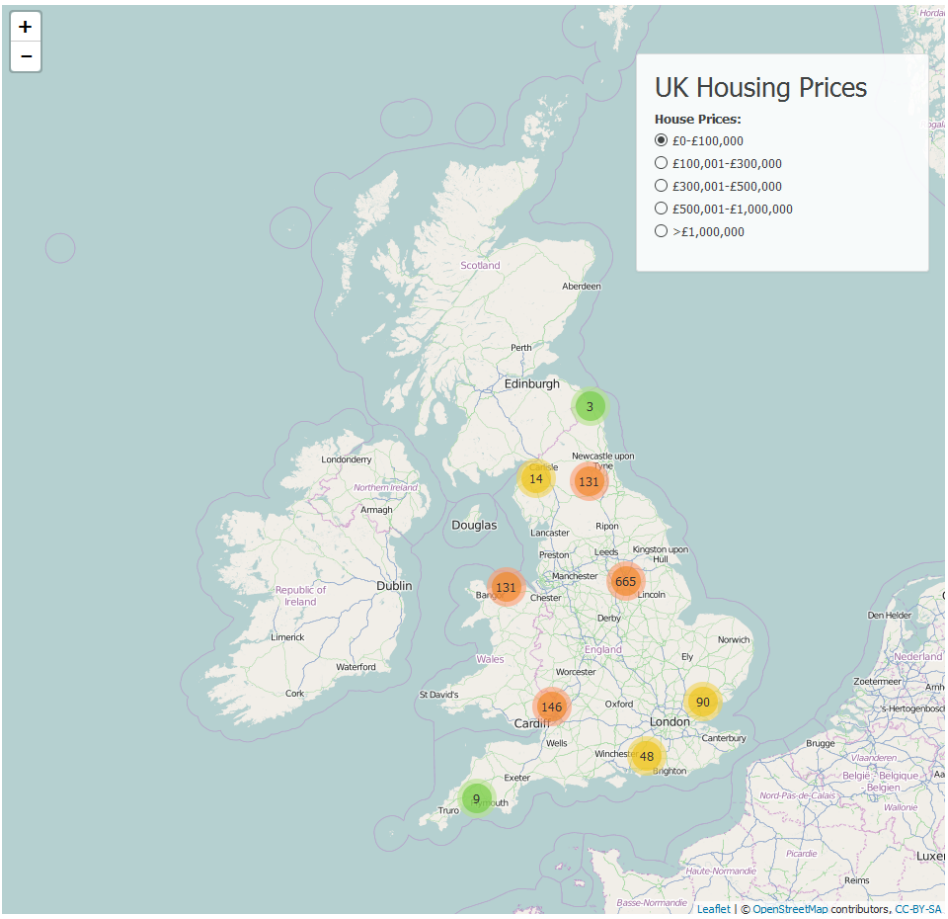
Analyzing Large Geographic Datasets using R Cluster Maps and MongoDB

Daniel Thorns

October 21, 2015



1 Introduction and Motivation

This is a proof of concept of potential use of R packages 'Leaflet' and 'RShiny' to create visual analytics tools which illustrate large geographic datasets. To prove this, a custom web application will be created to plot past house sales in the UK. This project also explores the performance benefits of holding data in a MongoDB database, compared against holding the data in a csv file.

2 Sourcing and cleaning the data

Files used:
Details about house sales (.csv):
Longitude/Latitude lookup (.csv):
Java code for writing the files: Available on the github page

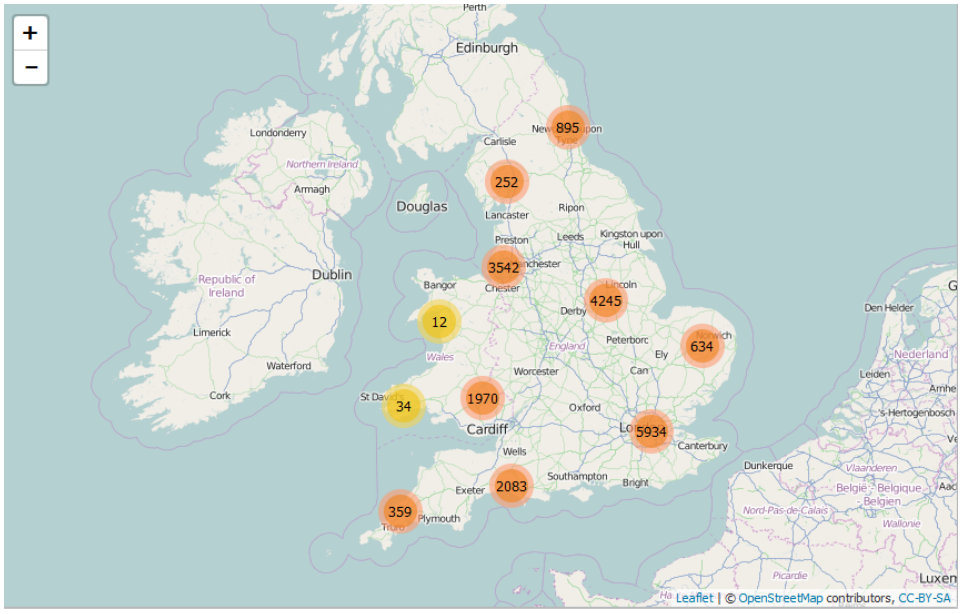
The house sales file contains details of house sales in the UK since the start of 2015 ($\sim$ 317,000 entries), with information about fields such as price, postcode, address and date of sale. However, to plot a map in R the longitude and latitude is needed (which is not included in this file), and so we use the Longitude/Latitude lookup file ($\sim$ 5,097,000 lines) to get this by using the postcode of the property. The java code mentioned above is used to find the information and write a new file.

There are two different variations of the java code, one is to write a .csv file and the other is to write the file in JSON format. The data needs to be in JSON format to be loaded into a NoSQL database, specifically MongoDB, which we will be using later when doing a comparison between the speed of loading data from a .csv file and a MongoDB instance.

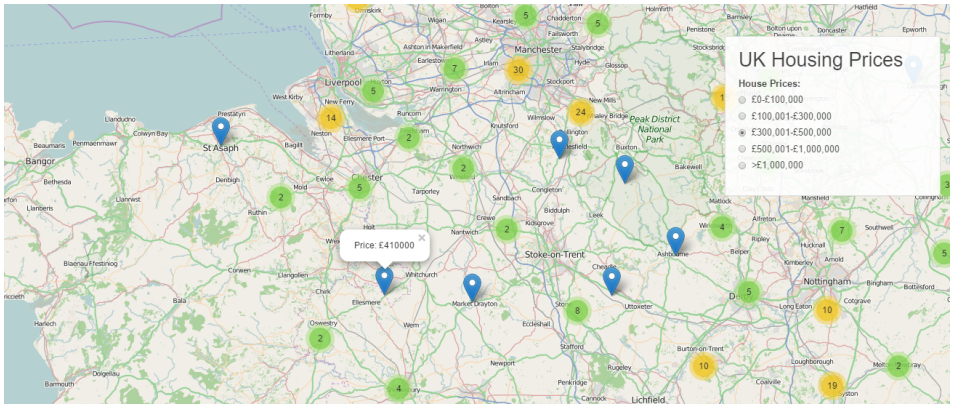
3 Drawing the initial map using the .csv file

Files used:
Code to plot cluster map: Available on the github page

The code mentioned above uses the Leaflet package to draw the map. In this example a set of 20000 markers are plotted. Here is an image of the map drawn:



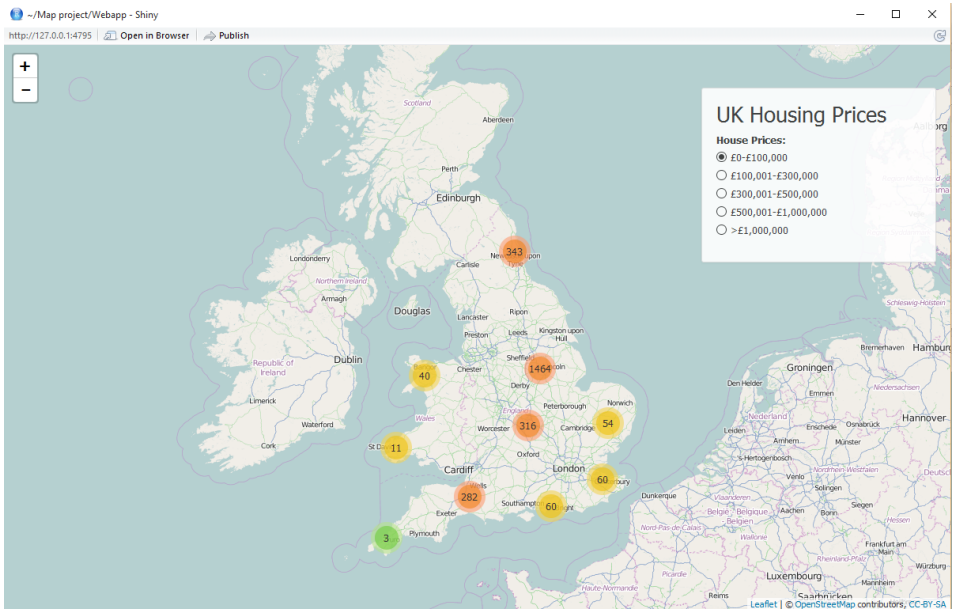
Leaflet allows high-quality maps to be drawn easily in R which are not just images of the area and instead allows zooming in and out, and the clusters dynamically reform depending on the zoom level.



4 Creating a web application and adding a user interface

Files used:
Code used to make web application: Available on the github page

The main motive behind creating a web application was to give the user the ability to select a house price range and then the map dynamically draw the points matching the critera. This was made possible with the RShiny package.



5 Changing from .csv file to MongoDB querying

Files used:
Web application with MongoDB: Available on github page
MongoDB setup: Available on github page

Instead of using a csv file to hold the data, it was loaded into a MongoDB database. This was done to determine which method of storing data yields the best performance.
This requires locally hosting a MongoDB database and connecting to it through the RMongo package. To perform this step, MongoDB needs to be installed on either the same computer or a database server. For my testing, MongoDB was installed on my windows pc. I have included the commands I used to set mine up on the github page.

6 Comparison tests

Files used:
Code used to compare speed: Available on github page

I compared the time taken to complete the following 2 ways of preparing the data for use with the web application:

1. Reading the .csv file into a data frame then subsetting the data according to the pricing critera
Code timed to execute:

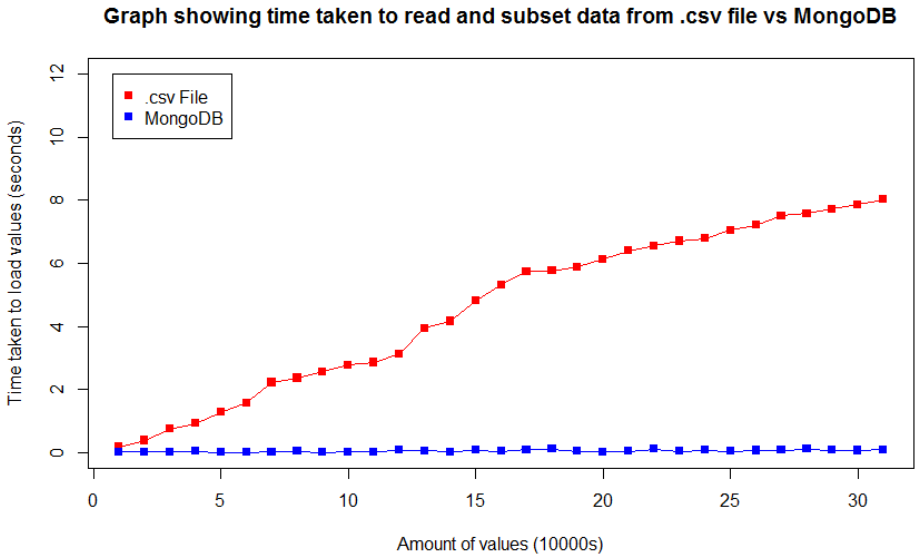
```
method1 <- function(j) {

  data <- read.csv('lonlatprice.csv', nrow = j)

  for(i in 1:ncol(data)) {
    data[,i] <- as.numeric(as.character(data[,i]))
  }

  data <- na.omit(data)
  colnames(data) <- c('longitude','latitude','price')

  d1 <- subset(data, data$price > 0 & data$price <= 100000)
  d2 <- subset(data, data$price > 100000 & data$price <= 300000)
  d3 <- subset(data, data$price > 300000 & data$price <= 500000)
  d4 <- subset(data, data$price > 500000 & data$price <= 1000000)
  d5 <- subset(data, data$price > 1000000)
}
```

Using this method, the values read in from the .csv have to have the type casted to a character and then to a number for R to acknowledge them as numbers. There is also another complication as the data has to have no nulls otherwise the application does not work, and so na.omit has to be used.

2. Querying the indexed MongoDB database using pricing criteria to return the relevant data
Code timed to execute:

```
method2 <- function(j) {
  test <- mongoDbConnect('hp')
  d1 <- dbGetQuery(test, 'houses', "{'price': {$gt: 0,$lt: 100000}}", limit = j/5)
  d2 <- dbGetQuery(test, 'houses', "{'price': {$gt: 100000,$lt: 300000}}", limit = j/5)
  d3 <- dbGetQuery(test, 'houses', "{'price': {$gt: 300000,$lt: 500000}}", limit = j/5)
  d4 <- dbGetQuery(test, 'houses', "{'price': {$gt: 500000,$lt: 1000000}}", limit = j/5)
  d5 <- dbGetQuery(test, 'houses', "{'price': {$gt: 1000000}}", limit = j/5)
}
```

Since there was 5 queries, I limited each query to $\frac{j}{5}$ entries, to ensure that the queries collectively returned j entries.



The graph shows that MongoDB is a much faster way of loading data compared to reading a .csv file, even for small datasets. When efficiency and speed are very important, such as for R web applications, loading the data this way can make the application run a lot smoother and faster.

7 How data is stored and retrieved with MongoDB

MongoDB stores its 'working set' in memory. The 'working set' comprises of both database indexes and associated data. Usually we use an index to find the location of the data that we want, and then do an additional database read to fetch that data. In our example however, all of the information that we are retrieving from MongoDB is held in the index (longitude, latitude, and price). As such our query to MongoDB is fully satisfied from the index, and there is no need for the additional operation to read data over and above what is in the index.

8 Improvements

Java reformatting: The code itself is inefficient as the method of looking up the respective longitude/latitude is not indexed,for each postcode the entire .csv

is itterated (from the start each time) until the postcode is found. This could be made quicker by reformatting and loading the lookup file into MongoDB, so it can be indexed and queries would be much faster.

System timing functions in R: For code cleanliness, I could have used the **foreach** package, which means that rather than having the code as follows:

```
calculateread <- function(t) {
b <- matrix(ncol = 3, nrow = t)
for (i in 1:t) {
  b[i,1] <- i
  b[i,2] <- system.time(method1(i*10000))[1]
  b[i,3] <-  system.time(method2(i*10000))[1]
}

return(b)
}
```

I could have done:

```
calculateread <- function(t) {
  return(foreach(i=1:t, .combine = 'rbind') %do% c(i,
  system.time(method1(10000*i))[1],system.time(method2(10000*i))[1]))
}
```

Which just makes the code much more compact.

9 Conclusion

Using MongoDB with Leaflet in R provides potential to gain quick locational analytics. In the example above house sales are used, but this could be anything where location is a relevant factor. For example, it could be used to visualise how location affects insurance claims, or to study how the life expectancy changes between lots of different areas.

This project was intended to be a proof of concept that loading large datasets into R is faster through database connection than reading a locally stored .csv file, however the potential for large scale locational data visualisation within R has also been shown.

We see better performance by reading data from MongoDB than from csv files. This is because with MongoDB, the information is held in memory instead of on disk. Memory gives much faster and consistent read access times than disk. Additionally, we see high performance because all of our required information can be held in an index, and as such MongoDB has no need to perform additional reads to retrieve data.