Karl Roth
Nuclear, Plasma, and Radiological Engineering (NPRE)
1828

GEOG 479 Midterm

SECTION 1 – CyberGIS

1. The key charactaristics of CyberGIS includes reproducibility, sustainability, usability, scalability, interoperability, and reliability. Reproducible means that if we use the same data and computational set up we should obtain the same results, this is important in all fields of science. Sustainability means that the system should be consistently accessible. Usability means that the system should be user-centric and focused on transparent access and ease of use. The system should also be high-perfromance and scalable, to accommodate for different number of users, quantities of data, and amount of applications that can be run on the system. Interoperability means that there should be a seamless integration between the components of the system as well as external processes. Finally, the system should be reliable. If the system has many bugs and lots of down time, it will no longer be usable and many participants will look elsewhere for their computing needs.
   There are many applications suitable to the CyberGIS framework. Including Climate change assessments, emergency management, geographic information science, geography and spatial sciences, geosciences, and social sciences. Additional communities include Biologists, Geographers, Geoscientists, Social Scientists, and the general public.
2. The three pillars of GIS are Cyberinfrastructure, GIScience, and Spatial Analysis and Modeling. Cyberinfrastructure consists of "computing systems, data storage systems, advanced instruments and data repositories, visualization environments, and people, all linked together by software and high-performance networks…" (Wang, Shaownen. CyberGIS software) Spatial Analysis and Modeling is the analysis of big spatial data and computationally intensive problems. GIScience, or Geographic Information Science, is the study of techniques to analyze, process, and visualize geospatial data.
3. The three computational modalities of ROGER are Batch System, Hadoop, and OpenStack Cloud. Batch System is equivalent to traditional high-performance computing. This is defined as the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop or workstation. The system can communicate either through shared memory or message passing between cores. Some possible applications include Airline ticket purchasing systems, Finance companies portfolio recommendation engines, and the simulation of astronomical and weather events. Hadoop is a specific frame work that works on a distributed file system, with many nodes and inherent redundancy. Some specific cases of Hadoop are MapReduce and Spark. The classic Hadoop example is the word count function, however it can be applied in any situation where the information is able to be distributed and operated on in parallel, before being aggregated during the reduce step. OpenStack Cloud is a cloud computing paradigm. Cloud Computing is computing in which large groups of remote servers are networked to allow the centralized data storage, and online access to computer services or resources. Clouds are great for storing data because of the redundant distributed nature of the system. This lends it to high reliability. Cloud computing is frequently used by companies like Google and Amazon, for they're advertisement targeting.

SECTION 2 – Geospatial Python

A pdf of the iPython notebook will be attached separately.

SECTION 3 – PostGIS and MongoDB
The code for this section will be included separately.
karoth4_SQL.py
karoth4_mongo.py

The data was put onto roger from the local computer using the scp command, which is essentially a combination between ssh and cp. The SQL script shown below in Figure 1 used psycopg2 to import the worldcities.csv into an SQL table with a POINT geometry object, and the mongo script used pymongo to import the worldcities.csv into a MongoDB table.

```
# Karl Roth
# Nuclear, Plasma, and Radiological Engineering (NPRE)
# 1828

import psycopg2

# Try to connect

try:
        conn=psycopg2.connect("dbname='midterm' user='g479' password='g479'")
        conn.autocommit = True
except:
        print "I am unable to connect to the database."

cur = conn.cursor()

cur.execute("""CREATE TABLE karoth4
                (
                 city varchar(40),
                 city_ascii varchar(40),
                 lat float,
                 lng float,
                 pop float,
                 country varchar(40),
                 iso2 varchar(6),
                 iso3 varchar(6),
                 province varchar(60)
                )""")


# After removing commas from all the strings the worldcities.csv header was
# removed and saved as worldcities_noheader.csv
f = open("worldcities_noheader.csv")
cur.copy_from(f,'karoth4',sep=",")
f.close()

cur.execute("""ALTER TABLE karoth4 ADD COLUMN geom geometry(POINT,4326)""")
cur.execute("""UPDATE karoth4 SET geom = ST_GeomFromText('POINT(' || lng || ' ' ||lat || ')',4326)""")
```

Figure 1: Screenshot of karoth4_SQL.py

Once the data was imported into the SQL table named ‘karoth4’ the number of cities in the data base was determined to be 7322 as shown in Figure 2.

```
midterm=> SELECT count(*) FROM karoth4;
 count
-------
  7322
(1 row)
```

Figure 2: SQL query on the karoth4 table to determine the number of cities.

Then the list of cities in the state of IL was determined using the query shown in Figure 3.

```
midterm=> SELECT city FROM karoth4 WHERE province ='Illinois';
      city
----------------
 Galesburg
 Joliet
 Cape Girardeau
 Rockford
 Evanston
 Rock Island
 Elgin
 Decatur
 Alton
 Quincy
 Urbana
 Bloomington
 Kankakee
 Waukegan
 Aurora
 Carbondale
 Belleville
 Springfield
 Peoria
 Chicago
(20 rows)
```

Figure 3: Demonstrates the query and results for all cities in Illinois.

Then a query was performed to find the number of cities within 500km of Urbana. The use of 5 distance units gave reasonable results that were comparable to the MongoDB query discussed later. When “500000” was used, the entire database was returned. There is clearly some error in the units.

```
midterm=> SELECT city FROM karoth4 WHERE ST_DWithin(geom, ST_GeomFromText(
'POINT(-88.204187 40.1099923)',4326),5);
midterm=> _
```

Figure 4: Query to determine the number of cities within 500km of Urbana. The results are shown below. This screenshot only captures the first cities.

```
      city
----------------
 Paragould
 Iowa City
 Ottumwa
 Kirksville
 Galesburg
 Joliet
 Cape Girardeau
 Rockford
 Evanston
 Rock Island
 Elgin
 Richmond
 Terre Haute
 Lafayette
 Marion
 South Bend
 New Albany
 Elkhart
 Hopkinsville
 Madisonville
 Bowling Green
 Springfield
 Columbia
 Sheboygan
 Waukesha
 La Crosse
 Tomah
 Janesville
 Appleton
 Benton Harbor
 Battle Creek
 Jonesboro
 Davenport
 Burlington
 Dubuque
 Waterloo
 St. Charles
 Poplar Bluff
 Columbia
 Decatur
 Alton
 Quincy
 Urbana
 Bloomington
 Kankakee
 Waukegan
 Aurora
 Carbondale
 Belleville
 Bloomington
 Muncie
 Kokomo
 Gary
 Fort Wayne
 Covington
 Bowling Green
 Paducah
 Owensboro
```

As for the MongoDB, the karoth4_mongo.py script was used to import the data from ROGER into MongoDB. This script is shown in Figure 5 below.

```
# Karl Roth
# Nuclear, Plasma, and Radiological Engineering (NPRE)
# 1828

from pymongo import MongoClient
import datetime
import json
import os
# The worldcities.csv file was cleaned to remove comma's from the strings
# this was neccesary for the csv to import properly.
def main():
        client = MongoClient('mongodb://localhost:27017')
        db = client.karoth4
        collection = db['cities']
        with open("worldcities.csv","rb") as mFile:
                next(mFile)
                for mLine in mFile:
                        [city,city_ascii,lat,lng,pop,country,iso2,iso3,province] = mLine.split(",")
                        latt = float(lat)
                        lngg = float(lng)
                        geoString = {'city': city, 'loc': {'type' : 'Point', 'coordinates' : [lngg, latt]}}
                        print geoString
                        db.cities.insert(geoString)
if __name__ == '__main__':
        main()
```

Figure 5: Screenshot of the karoht4_mongo.py script

Then a connection to monog was opened and the karoth4 database was used. The show command was performed to check if the cities collection had properly imported. This is shown in Figure 6 below. Then in Figure 7, the first entry of the table was displayed.

```
karoth4@postgres-g479:~$ mongo
MongoDB shell version: 2.6.10
connecting to: test
> use karoth4
switched to db karoth4
> show collections
chicago
cities
class
system.indexes
> _
```

Figure 6: Screenshot of opening the mongo database and viewing the collections. This figure demonstrates that the cities table did get imported.

```
> db.cities.findOne()
{
        "_id" : ObjectId("59f9f1f69a659f50b029b917")
,
        "city" : "Qal eh-ye Now",
        "loc" : {
                "type" : "Point",
                "coordinates" : [
                        63.13329964,
                        34.98300013
                ]
        }
}
```

Figure 7: Displaying the first entry of the cities table. It imported properly.

Next an index was created so that we could properly perform the geospatial query. Since we are looking for points within a distance, the "2dsphere" was chosen. This can be seen in Figure 8. Figure 9 demonstrates the actual query. Only the first data point was included in the screen capture for succinctness.

```
> db.cities.createIndex({loc:"2dsphere"})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
```

Figure 8: Creating the index for the geospatial query.

```
} db.cities.find({loc:{$nearSphere:{$geometry:{type:"Point",coordinates:[-88.204187,40.1099
923]},$minDistance:0,$maxDistance:500000}}}).sort({"loc":1})
{ "_id" : ObjectId("59f9f1fb9a659f50b029d488"), "city" : "Des Moines", "loc" : { "type" : "
Point", "coordinates" : [ -93.61998092, 41.57998008 ] } }
```

Figure 9: The actual query of data within 500km of Urbana. The urbana location was looked up in the worldcities.csv and entered manually.