OpenStreeMap Analysis Using MongoDB
Los Angeles, California

Joby John

April 25, 2016

Audit and Cleanup The los-angeles_California.osm file is **1.9GB** in size. The initial task is to read the elements of the xml task and convert the elements in to json elements and produce a json file as output. Large files like the osm file for Los Angeles are more than 1 GB in size and pose a memory problem while using the elementTree.iterparse() method. In order to iteratively parse the file, it was important to clear each element after it was processed using the **element.clear()** function.

Using *tags.py*, we find: {'lower': 3069661, 'lower_colon': 3020030, 'other': 197944, 'problemchars': 9} Among the problem tags, most "problems" were the use of space or . (period),; (semi colon).

Using *map_parser.py* Parsing the osm file we find the different kinds of tags in the file. {'bounds': 1,
'member': 84031,
'nd': 9439779,
'node': 7946023,
'relation': 11230,
'root': 1,
'tag': 6287644,
'way': 940999}

Using the `python #pymongo db.command("collstats",'losangeles1')` we find:
{u'avgObjSize': 259.30946115078103,
u'count': 8886382,
u'ns': u'cities.losangeles1',
u'size': 2304322928.0,
u'storageSize': 3210674176.0,
}

Counting the number of users, we find that there are about 3222 unique users (pipeline8) that created the data for Los Angeles. However, in curating the address field it was noticed that many of the *addr:street* have only a name and are without a proper ending identifying the *type* of way. These typeless streets are maintained as is. Some nodes and ways are even from another country (Austria). This has to taken care of during further wrangling with mongoDB.

Not only are there documents from other cities but also, one should be careful *not* to $match using the city name as the Los Angeles region consists of many smaller cities. The dataset is also corrupted by data from other states (within the US). It now becomes important to ensure that the data for Los Angeles reflects only local data. One way to ensure this would be to check for **Latitude** and **Longitude** coordinates.

The region of interest is found using the following tags, which are found in the beginning of the file: **minlat**: 33.298, **maxlon**: -116.724, **minlon**: -119.437, **maxlat**: 34.583

Only those documents (with 'type':'node') that lie in the area determined by the above coordinates are retained. This is addressed programatically in python (*data_json.py*) as we convert to json.

However, only documents of type 'node' have latitude and longitude specified. So documents of type 'way' could not be curated in this way. To overcome the problem of 'way' tags that do not lie in the LA region and do not have logitude and latitude fields a new method is used. This is described below in the MongoDB section.

The output json file so produced has a size of **2.1 GB**.

Some findings: There are many 'is_in:city','is_in:state_code','is_in:county','is_in:country', tags which are valid (i.e. no problem characters). It is possible to use these documents and introduce them to the address field. But this was not undertaken in the current verstion. Right now, these records are included as-is.

Also, importantly, none of the non "tag" tags have problem characters.

However, for the final conversion of the osm to json file, a naive approach that used only **element.iterparse()** and **element.clear()** proved to be insufficient. Therefore, a separate function **get_element()** as suggested here (http://effbot.org/elementtree/iterparse.htm) was included to process the osm file.

MongoDB

We list the number of cities with the following pipeline:

```
# Count all the cities
pipeline1 = [{"$match":{"address.city":{"$exists":True}}},
            {"$group":{'_id':'$address.city',"count":{"$sum":1}}},
            {"$group":{'_id':None,"count":{"$sum":1}}}]
```

By restricting the nodes and ways to the LA region's longitude and latitude range, the number of unique cities reduced from 362 to 361. Even this is not completely accurate list of cities. For example, there are spurious cities like "Ventura County" and "Newport Coast". These can be eliminated by checking against a list of legitimate California cities. Interestingly the LA OSM file had many more spurious cities (705) just a few weeks ago (cities from other countries and states). Re-downloading the file, the new version seems to be cleaner.

However, as mentioned previously, only "node" type documents have the latitude and longitude fields. The 'way' type documents do not have that field and hence we cannot weed spurious "way" documents.

In order to overcome this problem, the following method is used. 1. Create a list of all the cities with type 'node' (These are cities that legitimately lie in the LA region as defined by the min and max latitude and longitude).
2. Create a list of all the cities with type 'way'. This list contains both cities in the LA region and spurious cities as we could not filter them during the json creation.
3. Take the **Set difference** of the above two lists to get an approximate list of all the spurious ways.
4. In the list of spurious lists so obtained, there are a few legitimate cities (of ways without corresponding nodes or just because of a formate difference or a typo). These legitimate ones can easily be weeded out manually to create a more accurate list of spurious entries. Let's call this list Way_NotLA (i.e. ways not in LA).
5. Now, using the **$nin** operator we can easily avoid the cities that are not in the LA region even for the 'way' nodes.

```
# sort cities with max entries
pipeline5 = [{"$match":{"address.city":{"$exists":True,'$nin':Way_NotLA}}},
            {"$group":{'_id':'$address.city',"count":{"$sum":1}}},
            {"$sort":{'count':-1}}]
```

Now we find that there are *332* unique cities: Of course there are typos and a few reundancies like 'Los Angeles','Los |Angeles', and 'los angeles, ca', Los Angeles-Venice, etc. Even so, this is a more representative filtering of the documents.

It is found by sorting the documents grouped by 'address.city' that the city *'Irvine'* has the most documents (24304).

```
# Zipcode with most entries
pipeline6 = [{"$match":{"address.postcode":{"$exists":True}}},
            {"$group":{'_id':'$address.postcode',"count":{"$sum":1}}},
            {"$sort":{'count':-1}},
            {"$limit":10}] # sort cities with max entries
```

The zipcode 92630 is found to have most mentions (15097 docs). The other top zipcodes are 92028 (Fallbrook), 92620(Irvine), 92860 (Norco), 92618(Irvine), 92602 (Irvine), 92672 (San Clemente).

```
#City with the most unique zipcodes
pipeline7 = [{"$match":{"address.postcode":{"$exists":True,
            "address.city":{"$exists":True,'$nin':Way_NotLA},
            "address.city":{"$exists":True}}},
            {"$group":{'_id':'$address.city',
            "postal_code":{'$addToSet':'$address.postcode'}}},
            {'$unwind':"$postal_code"},
            {'$group':{"_id":'$_id','count':{'$sum':1}}},
            {'$sort':{'count':-1}},
            {"$limit":10}
            ]
```

By using the *$unwind* operator we can then *$addToSet* and find the number of unique postal codes for a give city. The city of Los Angeles come out at the top with 104 postal codes. Irvine (18), Long Beach (13), Riverside (12), Pasadena (11) are the other cities with most zipcode (top 5).

```
# User with most entry:
pipeline8 = [{"$group":{'_id':'$created.user','count':{'$sum':1}}},
            {'$sort':{'count':-1}},
            {"$limit":10}
            ] # The user with most entries
```

We find that in the sheer number of documents created, 'woodpeck_fixbot' leads the way, closely followed by Temecula_Mapper. AM909, kingrollo_imports,N76_import, nmixterare the other top contributors with around or more than quarter million documents each.

However if we restrict attention to cities with city names or even postcodes (i.e. look at users who have contributed to most cities or most postal codes), woodpeck_fixbot is not in the top 50 even.

```
# Users with contribution in most postal code.
pipeline10 = [{'$match':{"address.postcode":{'$exists':True}}},
            {"$group":{'_id':'$created.user',
            "postal_code":{'$addToSet':'$address.postcode'}}},
            {'$unwind':'$postal_code'},
            {'$group':{'_id':'$_id','count':{'$sum':1}}},
            {'$sort':{'count':-1}},
            {"$limit":90}
            ]
```

According to the above query pipeline: AM909, Brian@Brea, palewire,nmixter, mmaxerickson are the top contributors by postal codes.

```
# Return all the restaurants
pipeline12 = [{'$match':{"amenity":{'$exists':True,'$in':['restaurant','Restaurant']}}}]
```

Number of restaurants in the LA region is *3131*.

Restaurants Near A Location: One can use geoIndexing to query the number of amenities in a region: for example, a very common query would be to find all the restaurants within x miles of a given location, e.g. Staple's Center.

Problems encountered in this approach: See pipeline13 and pipeline14:

```
pipeline13 =   [{'$geoNear':{'near':[ -118.26684, 34.04302 ],'spherical':True,
'maxDistance':10/3963.2,'distanceField':'distance','includeLocs':'pos',
'distanceMultiplier':3963.2,'spherical':True,'limit':200000,'query':pipeline12}}]
```

In the above query 3963.2 is tha radius of the earth and is used to convert 'maxDistance' to radians $(d = r\theta \ \Rightarrow \ \theta = d/r)$
The last item in the query specifies pipeline12 which is a query of documents with amenity==restaurant. However, pipeline13 returns documents that do not meet this constraint. It seems that *query=pipeline13 does nothing* It returns $2e5$ documents in the vicinity of Staple's Center. We then use another query to extend this pipeline to return only restaurants.

```
pipeline14 = [{'$geoNear':{'near':[ -118.26684, 34.04302 ],
                           'spherical':True,'maxDistance':10/3963.2,
                          'distanceField':'distance','includeLocs':'pos',
                          'distanceMultiplier':3963.2,'limit':50000}},
          {'$match':{"amenity":{'$exists':True,$in':['restaurant','Restaurant']}}},
          {'$project':{'_id':'$_id',"amenity":"$amenity",
                           'name':'$name','dist':"$distance",'pos':'$pos'}}]
```

The above pipeline has a flaw that the limit is applied to all the results and the resulting collection is then queried with a $match. If we are near a high density point (like Staple's Center in downtown LA) then the $geoNear query might have more than 50000 results and therefore we might not be capturing all the valid restaurants.

To overcome this, a plain db.command() is used as follows:

```
result =db.command('geoNear','losangeles1',
                   near={'type':'Point',
                         'coordinates':[-118.26684, 34.04302]},
                 spherical=True,
                 maxDistance=333, # Around 1000 feet.
```

```
                  distanceField='distance',
                  includeLocs='pos',
                  query={"amenity":{'$in':['restaurant']}})
```

We find there are only four restaurants within 1000 feet (~330 m) of the Staple's Center.By adding the following to the query

```
query={"amenity":{'$in':['restaurant']},"cuisine":{'$regex':'.vegan.'}}
```

we find that there is only a single restaurant (Souplantation) that offers vegan options in the LA region. Another query on the name reveals that there are 3 Souplantations in the region.

Other ideas For Dataset: The audit process on the streetnames makes sure that the street names follow a certain convention. A similar approach can be applied to clean up the city names. Right now city names were partially curated by checking against the list of city names on nodes in the LA region. We can take a more aggressive approach by making sure that the city names do not have the state's name along with it ('CA', 'ca') by checking pruning the last characters.

A similar check can be applied for postal codes to follow the 5 digit pattern. Also, they can also be checked against a list of valid postal codes that belong to the LA region. This list of valid zipcodes could also be generated from the nodes whos coordinates lie within the latitude and longitude of the region.

Using k-means clustering:
K-means clustering can be used to identiy clusters of nodes that people are interested in, in a given city. By selecting say 20-30 different clusters for a region like LA, we can classify the nodes/ways in to different clusters. By idenitifying the clusters with most points, one can discover trends of "interesting areas" and guide users to regions with less points to increase use contribution in those areas. One would plot a color-coded cluster map of nodes on a map of LA to visualize the clustering of points. The color code would correspong to cluster number that each node belongs to. Psuedo-code for this is presented in *psuedo_code_kMeans.py*.

File Names

- *tags.py* Counts all the tags in the OSM file.
- *Users.py* Counts the number of unique users (3222) in the OSM file and returns their UIDs.
- *Audit.py* Counts the number of unique street types in the OSM file. Also counts the number of city names in OSM file.
- *data_json.py* Converts the OSM file to json. For tags of type "node" with a latitude and longitude, determines if it belongs to the LA region or not. Weeds out the documents with addr:city not in the LA region. Relies on curate_city_way_node.py to get the list Way_NotLA (which is the list of cities not in LA.)
- *city_way_node.py* This documents collects the result of two different pipelines of mongoDB query and takes the difference of the cities in "nodes" and cities in "ways" to come up with the cities not in LA region and stores it in the list Way_NotLA

- *Mongo_aggregate1.py* The main script that runs different queries on the collection 'losangeles1' in the database 'cities' These pipelines have been discussed in the pdf document.
- *psuedo_code_kMeans.py* Pythonish psuedocode describing how one could find clustering within the map data.
- *resize_OSM.py* Script to downsize the osm to a smaller size.