P2: Data Wrangling With MongoDB

By MOHAMMED MAHDI AKKOUH

Introduction

The area chosen here from OpenStreetMap is Singapore.
Singapore is a city-state in South-East Asia. It's a melting pot of many human kinds, and its cuisine reflects this: restaurants are everywhere. However, most of them do not have toilets, depending upon the public service.
The purpose of this analyses is to help find the nearest toilets to restaurants.

Important notice: the code enclosed with this document has been written using Python 3 and MongoDB 2.6.7; some code has been written with MongoDB shell version.

We'll start this analysis by auditing data and clean it. Then we'll propose a model to store in MongoDB and process it into MongoDB. Finally, we'll do some cleaning again and querying.

Auditing Data

First of all, we need to understand our data: how it looks like, what does it contain.

```
$> python3
>>> audit_counter()
{
  "bounds": 1,
  "node": 820540,
  "tag": 389043,
  "nd": 1012810,
  "relation": 1520,
  "member": 65316,
  "way": 123469,
  "osm": 1
}
```

Number of occurrence of each tag, file: src/audit.py

"bounds" and "osm" are just meta-data that we will ignore. The following functions display the relationships between all tags and their attributes:

```
>>> audit_tag()
{
 "relation": [
  "member",
  "tag"
 ],
 "way": [
  "nd",
  "tag"
 ],
 "node": [
  "tag"
 ],
 "osm": [
  "bounds",
  "relation",
  "way",
  "node"
 ]
}
>>> audit_attrib()
...
```

Relationships between tags and their content, file: src/audit.py

We understand that:

- “relation” contains multiple “member” that refers to a particular tag
- “way” contains “nd” that refers to a “node”
- “node” contains geographical coordinates

After looking directly to the OSM file, it seems that “tag” is the nucleon that contains many useful information in form key-value. By further auditing this tag, we find 1033 different keys and 563 base keys (without colons or before colons). Following is the TOP 10 of most used base keys:

```
>>> audit_key_counter()
[('highway', 69685),
 ('name', 54747),
 ('building', 50285),
 ('addr', 46380),
 ('source', 34826),
 ('oneway', 13535),
 ('garmin_type', 10120),
```

```
('created_by', 8234),
('catmp-RoadID', 7875),
('amenity', 6928)]
```

TOP 10 keys for “tag”, file: src/audit.py

And this is the list of all base keys:

```
>>> audit_key_colon_counter().keys()
dict_keys(['wetap', 'turn', 'old_name', 'maxheight', 'fuel', 'social_facility', 'gns', 'theatre',
'source', 'contact', 'official_name', 'payment', 'tunnel', 'service', 'tiger', 'wikipedia', 'alt_name',
'waterway', 'naptan', 'fire_hydrant', 'disused', '3dr', 'internet_access', 'abandoned', 'diet', 'addr',
'capacity', 'building', 'toilets', 'mtb', 'seamark', 'roof', 'memorial', 'destination', 'tower',
'communication', 'is_in', 'generator', 'name', 'public_transport', 'wheelchair', 'ele', 'ISO3166-1',
'traffic_signals'])
```

“tag” keys with colons, file: src/audit.py

“Toilet” appears in the list; Unfortunately, there are only 3 tags with this key. Looking again directly into the OSM file, we find that most toilets fall under the key “amenity”, so are restaurants.

Let’s move now to addresses and post codes. These information fall under the following keys: “addr:street” and “addr:postcode”. There are 11796 streets and 4357 post codes available.

Cleaning Data

Streets and post codes should respect some conventions. For example, the street type should not be abbreviated and should be available in one language, post codes should be a 6-digit code.

Problems with the dataset

- Abbreviated street types
- Some street types need translation
- Some addresses end with the house number
- Some addresses end with a direction qualifier: Admiralty Road West
- Some post codes are not 6-digit code

We’ll try to solve those problems with RE as detailed in the 2 following figures.

Solving address problems

After listing all street types available by extracting the last word of each address or sometimes looking directly into data, we end up with a list of valid street types and a mapping of incorrect street types. Then, we build a dictionary for each street/postcode with a status (whether we succeeded or not), the old value and the new value if available. Also, if the house number if available we keep it.
Following is the detailed process:

- Grouping all non-compliant addresses in a list
- Abbreviated street types are extracted from the end of the address and mapped to the correct type
- Identifying non-translated street types and mapping them with the appropriate translation
- Checking numbers at the end of addresses and extracting them

As already noticed, some parts of this process needed a quick look tat the original dataset in order to find an appropriate automated solution.

```
>>> audit_addr_street()

# ending with numbers: "Neo Tiew Lane 3"
ends_with_d_re = re.compile(r'[#\-\d]+\.?$')
# ending with non-white space characters:
ends_with_nonws_re = re.compile(r'\b\S+\.?$', re.IGNORECASE)
# extracting last word: "Neo Tiew (Lane) (3)"
get_last_w_and_d_re = re.compile(r'\s?(\w+)\s([#\-\d]+)\.?$', re.IGNORECASE)
# special translation
translate_re = re.compile(r'.*\b(jalan|lorong|jln\.?|jl\.?)\s(.+)$', re.IGNORECASE)
# street type is not the last word, but step-last
get_2_last_w_re = re.compile(r'\s?(\w+)\s(\w+)\.?$', re.IGNORECASE)
```

RE for cleaning addresses, file: src/audit.py

For example, this address “Ang Mo Kio Avenue 10” has been transformed to “10 Ang Mo Kio Avenue” and the house number added with the value “10”. On the other hand, this address “Mukakuning Indah 1 Blok U” has been rejected.

Solving post code problems

Singapore residential postal code is a 6-digit number, sometimes 5 but this means that the first digit is 0. This number allow to identify each building in the City. It is an alternative to the address text.
After cleaning, we end up with 3808 valid and unique post codes, and 8 non valid.
Following is the detailed process:

- Checking for non compliant post code numbers
- For 5-digit codes, adding a 0 at the beginning

- For less, ignoring

```
>>> audit_addr_postcode()

# 5-digits post codes
with_5_d_re = re.compile(r'[^\d]*([\d]{5})$')
# 6-digits post codes
with_7_d_re = re.compile(r'[^\d]*([\d]{6})$')
```

RE for cleaning post codes, file: src/audit.py

For example, this post code “18961” has been transformed to “018961”, and this one “135” has been rejected.

Modeling and Processing Data

We’ll use the same model as “Lesson 6”.
Processing is done in this call:

```
>>> process_map(True)
```

Processing data, file: src/process.py

And exporting to MongoDB in this call:

```
$> mongoimport --jsonArray -d db_name -c collection_name --file ../data/singapore.json
```

Exporting data in MongoDB, file: src/process.py

The option --jsonArray was necessary because the input file is huge.
Following some stats:

```
OSM file: 175MB
JSON file: 247MB
DB: 0.453GB

def most_active_users(db):
    pipeline = [ { "$group" : { "_id" : "$created.user",
                                "count" : { "$sum" : 1 } } },
                 { "$sort" : { "count" : -1 } },
                 { "$limit" : 10 } ]
    return db.singapore.aggregate(pipeline)
```

```
>>> most_active_users(db)
Number of users: 1036

def number_of_tags(db):
    pipeline = [ { "$group" : { "_id" : "$type",
                                "count" : { "$sum" : 1 } } } ]
    return db.singapore.aggregate(pipeline)
>>> number_of_tags(db)
Number of nodes and ways: 820534 / 123469
```

Stats, file query.py

We can notice that we've lost in our processing only 6 nodes (see number_of_tags() in file query.py).

Querying Data

Indexing

At the beginning, we'll create an index on "pos" and "amenity".
We've chosen the GeoJSON model for "pos", so we had to change "pos" type from [lon, lat] to { "type" : "Point", "coordinates" : [lat, lon] }

```
def update_pos(db):
    spore = db.singapore
    for doc in spore.find():
        if (type(doc["pos"]) != list):
            continue

        spore.update({ "_id" : doc["_id"] },
                    { "$set" : {
                        "pos" : {
                            "type" : "Point",
                            "coordinates" : doc["pos"]
                        }
                    } })
```

Updating "pos" to "2dsphere", file src/query.py

But, creating an index on "pos" resulted in an error because those coordinates were not stored in the expected order (longitude then latitude). Another error occurred because some coordinates were nil, so we've got rid of them. Running the command was successful after that.

```
spore.createIndex( { "pos" : "2dsphere"} )
spore.createIndex( { "amenity" : 1 } )
```

Indexing

After indexing, the DB size jumped to 0.953GB.

Querying

TOP 10 of most active users:

```
>>> client = MongoClient('localhost:27017')
>>> db = client.p2_wrangling_data
>>> most_active_users(db)
[{'_id': 'JaLooNz', 'count': 241662},
 {'_id': 'cboothroyd', 'count': 55177},
 {'_id': 'rene78', 'count': 51637},
 {'_id': 'kingrollo', 'count': 37622},
 {'_id': 'jaredc', 'count': 36204},
 {'_id': 'Sihabul Milah', 'count': 30562},
 {'_id': 'zomgvivian', 'count': 21389},
 {'_id': 'matx17', 'count': 20739},
 {'_id': 'singastreet', 'count': 19289},
 {'_id': 'fusionstream', 'count': 18862}]
```

TOP 10 users

Let's extract available toilets and restaurants:

```
$> mongo
> spore = p2_wrangling_data.singapore
> toilets = spore.find({ "amenity" : { $regex : /toilet/, $options : 'i' } })
> toilets.count()
136
> restos = spore.find({ "amenity" : { $regex : /restaurant/, $options : 'i' } })
> restos.count()
625
```

All restaurants and all toilets

Let's pick at random a restaurant and query the 10 nearest toilets with a maximum distance of 10km.

```
>>> resto = restos[100]
```

```
{'_id': ObjectId('558d73e6ce8e45de22aa7554'),
 'amenity': 'restaurant',
 'created': {'changeset': '5962754',
             'timestamp': '2010-10-05T16:34:16Z',
             'uid': '194337',
             'user': 'Sihabul Milah',
             'version': '2'},
 'garmin_type': '0x2a12',
 'id': '902904798',
 'name': 'Pak Datok',
 'pos': {'coordinates': [104.03768, 1.12966], 'type': 'Point'},
 'source': 'Batam Mapping Project',
 'type': 'node'}
>>> nearest_toilet(db, resto["_id"], 10000)
{
  "source": "Batam Mapping Project",
  "_id": "558d7386ce8e45de22aa29ee",
  "id": "719058594",
  "created": {
    "timestamp": "2011-01-02T01:01:05Z",
    "uid": "371324",
    "version": "2",
    "changeset": "6833952",
    "user": "Rudi Heitbaum"
  },
  "name": "M3 Building",
  "type": "node",
  "garmin_type": "0x2f10",
  "pos": {
    "type": "Point",
    "coordinates": [
      104.01645,
      1.12867
    ]
  },
  "amenity": "toilets"
}
...
```

Nearest toilet, file src/query.py

Discussion

What other information could we extract from this data? We can for example query the most dense area of whatever amenity. To do this, we should be able to retrieve maximums/minimums coordinates for the area, split it in small rectangles, and query within each small area.
I've started by getting the maximums/minimums but was limited by the aggregation pipeline. Indeed, I couldn't filter only one element of the list "coordinates".

```
# extreme coordinates
def extrems(db, min_or_max, lon_or_lat):
    if min_or_max not in ["min", "max"]:
        return None
    if lon_or_lat not in ["longitude", "latitude"]:
        return None
    minmax = -1
    if min_or_max == "min":
        minmax = 1
    lonlat = 0
    if lon_or_lat == "latitude":
        lonlat = 1
    pipeline = [ { "$project" : { lon_or_lat : "$pos.coordinates[{0}]".format(str(lonlat)) } },
                 { "$sort" : { lon_or_lat : minmax } },
                 { "$limit" : 1 } ]
    return db.singapore.aggregate(pipeline)
```

Getting extreme latitudes/longitudes, file query.py

Beyond this technical limitation, arises a topological limitation. Querying inside small rectangles returns the most dense rectangle but not necessarily the most dense area. Indeed, density is the number of units divided by the surface. As we have more units in an area, its density will increase, but increasing the surface without including enough units will reduce the density. The challenge is to find the most small area having the most number of units. So we need more flexibility on the shape of the area. A potential solution will be to split the area in very small rectangles and group them given their density and distance with an optimization function. Marketing companies could benefit from this information. Given their market segment, they can query for the most dense areas where they can approach their targets customers, hence optimizing their costs.