Project 3: Data Wrangling with MongoDB

Table of Contents

Intro....1
Getting Familiar with the Dataset....1
Summary of Issues....6
Data Processing....6
Additional Ideas....8
More Analysis....8
Conslusion....9

Intro

This document contains my answer to the Project 3: Data Wrangling with MongoDB.

While making my response I used the Udacity course materials mostly. In some cases I googled for solutions and went to StackOverflow.com and official MongoDB documentation. No other materials were used.

I include a few lines of code below, mainly for MongoDB queries used for the data analysis. Since the output sometimes is really huge, I limited the output to reasonable amounts to make it readable for my reviewer. In the real investigation, I didn't use any limits.

Getting Familiar with the Dataset

For my project I exported a piece of the map of Moscow, Russia. Here is the link to the map I used in my project:

http://overpass-api.de/api/map?bbox=37.4417,55.6435,37.6765,55.7331

The reason I selected this area is I have been living here all my life and I know this are quite well.

First I made a very basic query just to see some data from the .osm file. Immediately I got an error:

UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-6: ordinal not in range(128)

It made me think there is something wrong the character encoding. I solved it by using the **.encode('utf-8')** method. This knowledge will be useful later as well.

Let's look at data considering Unicode:

```
alt_name: Кунцево
name: Кунцево I
railway: station
highway: traffic_signals
colour: orange
name: Шаболовская
name:en: Shabolovskaya
network: Московский метрополитен
<Output cut>
```

Now let's count different tags that exist in the data.

```
'name': 8,
 'name:cs': 2,
 'name:en': 6,
 'name:fr': 2,
 'name:it': 2,
 'name:pl': 2,
 'name:pt': 2,
 'name:sv': 2,
<Output cut>
```

I immediately noticed that there are hundreds of different tags: with one colon, with two colons. There are also obviously similar items such as "color" and "colour". There are dozens of "name" tags probably with international notations. It can take a lot of time to fully clean even this small piece of the Moscow map. Let me have a look at more details and then I'll make certain decisions on which cases to solve.

I also noticed that there are lots of different address-related entries. Let's have a look at tags having an "address" attribute first:

```
===New Item===
address: Мичуринский пр-т, д. 20, к. 1 (Олимпийская деревня)
<Output cut>
```

Ok, these lines sometimes have the whole address in one line:

StreetName St., h. HouseNumber, bld. BuildingNumber

This issue I will fix. I will parse such entries using the *split_address_line()* function.

Now let's look at weird entries having “addr” and “addr2” entries:

```
===New Addr2 Item===
building: apartments
addr:housenumber: 6
addr2:street: Фруктовая улица
addr2:housenumber: 17
cladr:code: 77000000000265100
building:levels: 5
addr:street: Симферопольский бульвар
<Output cut>
```

Let's also look at the entries having both ‘addr:housenumber’ and 'addr:housenumber2' tags:

```
===New Addr2 Item===
addr:housenumber: 12 к2
building: yes
building:levels: 5
addr:housenumber2: 12 к2 с5
addr:street: улица Доватора
<Output cut>
```

Okay, I understood what we have here! In Russia, when a house sits at the cross of two roads, it has the number of X/Y, where X is the house number at Road 1 and Y is the house number at Road 2. OSM users sometimes add such info by making two address entries: 'addr' - Road 1 name and house number by Road 1 and 'addr2' - Road 2 name and house number by Road 2. House numbers can still be in the X/Y format. I will keep this format since it's official, but I decided to create a "secondary_address" entry for addr2 and similar entries.

The same idea is sometimes implemented as ‘addr:street2’ and ‘addr:housenumber2’. I will take care of this case as well.

Let's have a look at the ‘fuel’ entries. I noticed that there are many tags starting with ‘fuel’. What’s up there?

```
fuel:diesel: yes
fuel:octane_95: yes
fuel:octane_98: yes
fuel:diesel: yes
fuel:lpg: no
fuel:octane_92: yes
```

<Output cut>

Ok, it seems like this is about fuel stations offering different sorts of petroleum. Let me transform this to the following structure:

fuel:octane_95: yes
fuel:octane_98: yes

will become

fuel: {
octane_95: yes,
octane_98: yes
}

The same logic applies for "payment" entries (example data not shown). I will transform it as follows:

payment:visa: yes
payment:mastercard: yes

will become

payment: {
visa: yes,
mastercard: yes
}

Finally let's have a look at the numerous 'building' entries:

{'building:color': 'grey'}

{'addr:housenumber': u'1 \u04411',

'addr:street': u'\u041a\u0430\u043d\u0430\u0442\u0447\u0438\u043a\u043e\u0432\u0441\u043a\u0438\u0439 \u043f\u0440\u043e\u0435\u0437\u0434',

'amenity': 'bank',

'building': 'yes',

'building:color': 'grey',

'building:levels': '2',

'operator': u'\u041e\u043a\u0435\u0430\u043d \u0411\u0430\u043d\u043a'}

{'building:color': '#E3FAFA'}

{'building': 'yes',

```
 'building:color': '#E3FAFA',
 'building:levels': '1'}
{'building:color': '#E3FAFA'}
<Output cut>
```

Ok, we have a complicated structure for tags that start with "building". I decided to go for the following transformation:

building: school
building:roof: concrete
building:roof:colour: #C7C7C7
building:levels: 1
building:levels:underground: 1

will become

```
building: {
        type: school,
        roof: {
                type: concrete,
                colour: #C7C7C7
                },
        levels: {
                amount: 1,
                underground: 1
                }
         }
```

Other key:value pairs will remain unchanged.

Also I noticed from the outputs above that there are many international options for "name", "alt_name" and "official_name" entries. Let me go for the following transformation:

```
name: NameInRussian
name:en: NameInEnglish
name:de: NameInGerman
```

will become

```
name: NameInRussian
name_international: {
                        en: NameInEnglish,
                        de: NameInGerman
                        }
```

Same for "alt_name" and "official_name"

That's it. There are much, MUCH more other things that could be taken care of. But it will take ages to go through them. Let me focus on the fixes that I selected above and implement them.

Summary of Issues

I decided to programmatically fix the following issues:

1. Nodes having a single ‘address’ line;
2. Nodes having ‘addr’ and ‘addr2’ and similar entries;
3. Nodes having multiple ‘fuel’ entries;
4. Nodes having multiple ‘payment’ entries;
5. Nodes having multiple ‘building’ entries;
6. Nodes having international options for ‘name’, ‘alt_name’ and ‘official_name’.

Next come functions for the transformation process. I copied the approach for "created" and "position" from the course.

Data Processing

Sizes of files that I work with:

```
File: part_of_moscow.json           Size: 97687326 bytes

File: part_of_moscow.osm            Size: 88206345 bytes
```

Ok, both files are more than 50 MB.

At this point, I load my JSON file into the MongoDB instance using the following OS command:

```
c:\Program Files\MongoDB\Server\3.2\bin>mongoimport -d examples -c moscow --file "d:\Udacity\P3 Data Wrangling with MongoDB\part_of_moscow.json"
2016-04-24T13:24:17.010+0300 connected to: localhost
2016-04-24T13:24:19.985+0300 [######..................] examples.moscow 23.6 MB/93.2 MB (25.3%)
2016-04-24T13:24:22.985+0300 [#############.............] examples.moscow 48.6 MB/93.2 MB (52.2%)
2016-04-24T13:24:25.985+0300 [###################.....] examples.moscow 72.5 MB/93.2 MB (77.8%)
2016-04-24T13:24:28.311+0300 [########################] examples.moscow 93.2 MB/93.2 MB (100.0%)
2016-04-24T13:24:28.312+0300 imported 412257 documents
c:\Program Files\MongoDB\Server\3.2\bin>
```

Now let's investigate data in the database with some queries. Number of documents in the database:

number_of_documents = db.moscow.find().count()

```
Number of Moscow documents:  412257
```

Ok, it's the same as the number of documents that were imported using mongoimport.

Number of 'node' and 'way' tags:

```
pipeline = [{ "$group" : { "_id" : "$type", \
                    "count" : { "$sum" : 1 } } }]
pp.pprint([doc for doc in db.moscow.aggregate(pipeline)], width=4)
```

```
[{u'_id': u'node',
  u'count': 348210},
 {u'_id': u'way',
  u'count': 64047}]
```

Let's see the number of unique contributors. I'm intentionally using a dirty way, but I didn't want to copy the nice query based on "distinct" from the submission example :)

```
pipeline = [{ "$group" : { "_id" : "Unique users", \
                    "unique_users" : { "$addToSet" : "$created.user"} } }, \
        { "$project" : { "Count" : { "$size" : "$unique_users" } } }]
pp.pprint([doc for doc in db.moscow.aggregate(pipeline)], width=4)
```

```
[{u'Count': 1012,
  u'_id': u'Unique users'}]
```

Top 10 contributing users. Note the Unicode characters in user names!

```
pipeline = [{ "$group" : { "_id" : "$created.user", \
                    "count" : { "$sum" : 1 } } }, \
        { "$sort" : { "count" : -1 } }, \
        { "$limit" : 10 }]
result = db.moscow.aggregate(pipeline)
for doc in result:
    print('User: {0}, Contributions: {1}'.format(doc['_id'].encode('utf-8'), doc['count']))
```

```
User: trolleway, Contributions: 31906
User: s-tikhomirov, Contributions: 24743
User: ad47, Contributions: 23915
User: Павел Гетманцев, Contributions: 18349
User: AMDmi3, Contributions: 16942
User: literan, Contributions: 16705
User: KekcuHa, Contributions: 13063
User: Felis Pimeja, Contributions: 12634
User: stlp, Contributions: 8886
User: SergZL, Contributions: 8573
```

Let's see how many documents have a "secondary_address" info that I created while shaping.

```
pipeline = [{ "$match" : { "secondary_address" : { "$exists" : 1 } } }, \
        { "$group" : { "_id" : "Documents having a secondary address", \
```

```
                    "count" : { "$sum" : 1 } } }]
pp.pprint([doc for doc in db.moscow.aggregate(pipeline)], width=4)

[{u'_id': u'Documents having a secondary address',
  u'count': 38}]
```

Additional Ideas.

I launched the iD browser in order to see how the mapping process looks like in reality. I noticed that users can create any tags and put any values when describing objects.

Two examples that I noticed while initially analyzing the data set:

Example 1. Some buildings have two tags: 'color' and 'colour'. I guess we are talking about same things here. Obviously, we have here a confusion caused by language nuances. The system might limit options of "colour"/"color" to just "color".

Also the "color" tag might have either literal values: grey, or hex values, such as "#E3FAFA". Why doesn't one limit this field to just one type of data?

Example 2. In Russia, houses can have "building numbers" and "house bodies". In fact, these are same things, but there are many cases when a house has a house number, a building number and a body number: StreetName St., 17 (house number), bld. 2 (building number), bd. 1 (body number). This is the official notation.

I noticed in the editor that in some cases the "House Number" entry contains values such as "17 bld2". It can be confusing when doing a search. Some people might search for "17 bld2", some people might search for "17, bld2", some for "17b2" etc.

I think map editors should enforce some policy against such fields. For example, if a user enters letters in the "House Number" field, he might receive a warning saying that unless the official house number includes letters, user should use other fields (such as body number, building number, etc.) to specify corresponding information.

I understand that my suggestions might contradict to the XML paradigm which is about any tags, any values. However, for the sake of data uniformity and clarity, one might think of it, as sometimes some items should conform to certain rules.

More Analysis

A couple of additional queries against the database.

Let's calculate different types of buildings that we have in the database:

```
pipeline = [{ "$match" : { "building.type" : { "$exists" : 1 } } }, \
        { "$group" : { "_id" : "$building.type", \
                    "count" : { "$sum" : 1 } } }, \
        { "$sort" : { "count" : -1 } }]
```

```
pp.pprint([doc for doc in db.moscow.aggregate(pipeline)], width=4)


[{u'_id': u'apartments',
  u'count': 3535},
 {u'_id': u'garages',
  u'count': 1309},
 {u'_id': u'school',
  u'count': 336},
 {u'_id': u'kindergarten',
  u'count': 214},
 {u'_id': u'commercial',
  u'count': 136},
<Output cut>
```

Finally let's see how many documents have a "name_international" info that I created while shaping. A bit more complex query than the one above...

```
pipeline = [{ "$match" : { "$or": [ \
                        { "name_international" : { "$exists" : 1 } }, \
                        { "alt_name_international" : { "$exists" : 1 } }, \
                        { "official_name_international" : { "$exists" : 1 } }, \
                        ] } }, \
        { "$group" : { "_id" : "Documents having international names", \
                "count" : { "$sum" : 1 } } }]
pp.pprint([doc for doc in db.moscow.aggregate(pipeline)], width=4)


[{u'_id': u'Documents having international names',
  u'count': 1218}]
```

Conslusion

Data for even a small piece of Moscow, Russia, are quite dirty. It will require a huge amount of time and, most important, many decisions on how to handle them. A lot of data will probably will have to be dropped and re-entered using a more unified data schema.

Now I fully understand why the task of Data Munging and Wrangling is so important and why it consumes a great deal of the Data Scientist's time.

I believe that even when one can use very flexible schemas it might make sense to enforce some policies against the data coming into these schema.