OpenStreetMap Data Case Study. Data Wrangling with MongoDB.

Map Area

Riga, Latvia

https://s3.amazonaws.com/metro-extracts.mapzen.com/riga_latvia.osm.bz2
(https://s3.amazonaws.com/metro-extracts.mapzen.com/riga_latvia.osm.bz2)

This is my hometown and I'm naturally quite curious in analyzing it.

Problems encountered in the map

1. Value and nested key conflicts in tags

The dataset contains tag keys that have both an explicit value and additional children.

```
<tag k="railway" v="switch"/>
<tag k="railway:switch" v="double_slip"/>
<tag k="railway:switch:electric" v="yes"/>
<tag k="railway:switch:configuration" v="inside"/>
```

Key `railway` in the resulting JSON object can't have both the value `switch` and its nested object with additional values (`electric` and `configuration`). At first it might seem that the value can be discarded because the value for `railway` and the key of its nested object match (i.e. `switch`), but it isn't the case for `railway:switch`.

I have decided to store children information in a nested object and store parent's value within the same object with the key `value`.

Example:

```
{
  railway: {
    value: 'switch',
    switch: {
      value: 'double_slip'
      electric: "yes",
      configuration: "inside"
    }
  }
}
```

Following code has converted the original XML to the resulting JSON:

```
d = node
keys = key.split(':')

#Reserved key for resolving nested object and explicit value conflicts
assert all(key != 'value' for key in keys)

for inner_key in keys[:-1]:
    if inner_key in d and type(d[inner_key]) != dict:
        old_value = d[inner_key]
        d[inner_key] = dict(value=old_value)
    elif inner_key not in d:
        d[inner_key] = dict()

    d = d[inner_key]

if keys[-1] in d:
    d[keys[-1]]['value'] = value
else:
    d[keys[-1]] = value
```

In order to safely introduce the reserved key `value` I have scanned the whole dataset to make sure that none of the keys use it. Following line has ensured that:

```
assert all(key != 'value' for key in keys)
```

Unfortunately, this means that Mongo queries need to consider both `railway` and `railway.value` values.

2. Typed values (integers, floats, booleans) are stored as strings

Following keys store integers, floats, and booleans:

- `maxspeed`
- `capacity`
- `frequency`
- `voltage`
- `layer`
- `lanes`
- `building:levels`
- `tracks`
- `gauge`

However, there are some special cases:

- `lanes` can be an integer, but also a `slight_left|slight_left;through|through|through`.

Strategy: if it's an integer - convert it, otherwise split the string by `|` and store the values as an array. There was one special case with value `2;1` that I have discarded because it doesn't fit the format of the dataset and won't be useful in querying.

- `capacity` can store approximate values such as `~50` or `3-4`, but also string values such as `daudz`, which translates to `a lot`. There're are only 3 cases like that and since these values aren't known, I have discarded them.
- `voltage` is mostly is an integer, but sometimes it specifies a list of integer such as `110000;20000;3000`. There was one special case with value `110000;fixme`, which I have transformed to `110000`.
- `building:levels` are mostly integers, but sometimes it is stored as float. There're also two special cases such as `2B` and `1;2` that don't make much sense - I have discarded them.

I have converted other values to corresponding types to enable queries such as counting all streets with max speed greater than 50km/h.

Following code converts the values and discards special cases:

```
class SpecialCaseException(Exception):
    pass

def convert_value_if_applicable(key, value):
    if key in INTEGER_KEYS:
        value = int(value)

    elif key == 'lanes':
        try:
            value = int(value)
        except:
            value = value.split('|')

    elif key == 'capacity':
        if value in ['~50', '3-4', 'daudz']:
            raise SpecialCaseException()
        else:
            value = int(value)

    elif key == 'voltage':
        if value == '110000;fixme':
            value = 110000
        else :
            try:
                value = int(value)
            except:
                value = map(int, value.split(';'))

    elif key == 'building:levels':
        if value in ['2B', '1;2']:
            raise SpecialCaseException()
        else:
            value = float(value)
```

```
    elif value == 'yes':
        value = True

    elif value == 'no':
        value = False

    return value
```

Postcodes

Standard way of representing postcodes in Latvia is `LV-XXXX`, where X can be any digit. Most postcodes in the dataset have correct format, but there are some entries that had to be cleaned, such as:

- LV-2137 Garkalne
- 1045
- LV - 3002
- LV2111
- LV-2128;LV-2112

Following regex pattern and python code were used to clean these values:

```
postcode_pattern = re.compile(r'(LV)*[-]?\s*(\d{4})')
```

```
def clean_postcode(postcode):
    matches = postcode_pattern.findall(postcode)
    postcodes = map(lambda x: 'LV-' + x[1], matches)

    if len(postcodes) == 1:
        return postcodes[0]
    elif len(postcodes) > 1:
        return postcodes
    else:
        return None

...

def shape_node_tags(element, node):

...
        if key.startswith('addr:'):
            key = key.replace('addr:', '')

            if ':' not in key:
                if 'address' not in node:
                    node['address'] = {}

                if key == 'postcode':
                    postcode = clean_postcode(value)
```

```
                    if postcode:
                        node['address']['postcode'] = postcode
                else:
                    node['address'][key] = value
```

Unicode characters

The dataset has a lot of unicode characters – mostly Latvian and Russian languages.

```
"ru": "\u0443\u043b\u0438\u0446\u0430 \u0410\u043a\u043c\u0435\u043d\u044e"
```

Fortunately, MongoDB handles them very well. No further actions are necessary.

Overview of the Data

File sizes

- riga_latvia.osm: 342.2 MB
- riga_latvia.osm.json: 392.7 MB

Number of documents

```
> db.riga.count()
1788247
```

Number of nodes

```
> db.riga.find({type: "node"}).count()
1561236
```

Number of ways

```
> db.riga.find({type: "way"}).count()
226589
```

Number of unique users

```
> db.riga.distinct("created.user").length
888
```

Top 5 contributing users

Query

```
db.riga.aggregate([{
```

```
    "$group": {
      "_id": "$created.user",
      "count": {
        "$sum": 1
      }
    }
  }, {
    "$sort": {
      "count": -1
    }
  }, {
    "$limit": 5
  }]);
```

Result

```
Raitisx: 321654
AkageMuk: 168569
Pecisk: 161366
extropy_terplans: 103783
iav: 95614
```

Other ideas about the dataset

Top 10 appearing amenities

Query

```
db.riga.aggregate([{
    "$match":{
      "amenity": {
        "$exists":1
      }
    }
  }, {
    "$group":{
      "_id": "$amenity",
      "count":{
        "$sum":1
      }
    }
  }, {
    "$sort":{
      "count":-1
    }
  }, {
    "$limit":10
  }])
```

Result

```
parking: 2475
bench: 469
atm: 390
cafe: 339
restaurant: 338
school: 274
fuel: 237
kindergarten: 214
pharmacy: 200
recycling: 196
```

The results suggest at least three use cases for the dataset:

- `parking` and `fuel` would be quite useful to applications used by drivers (mobile, car built-in GPS navigators etc.).
- `atm`, `cafe`,`restaurant`, `pharmacy` are especially useful for applications used by tourists and locals
- `school`, `kindergarten`, `pharmacy` can be used for providing information to locals, especially families.

Top 10 tourism amenities

Query

```
db.riga.aggregate([{
  "$match":{
    "tourism": {
      "$exists":1
    }
  }
}, {
  "$group":{
    "_id": "$tourism",
    "count":{
      "$sum":1
    }
  }
}, {
  "$sort":{
    "count":-1
  }
}, {
  "$limit":10
}])
```

Result

```
hotel: 137
```

```
information: 63
museum: 55
viewpoint: 43
guest_house: 35
hostel: 33
attraction: 30
artwork: 28
picnic_site: 24
camp_site: 22
```

Knowing the city, I can state that there is clearly not enough data on tourist amenities.

Conclusion

It was very interesting to clean and explore the dataset of my hometown. It is clear to me that the data is incomplete as it is missing some amenities as well as in-depth information about them (such as opening times, cuisine information for restaurants, type of art etc.). Nevertheless, I'm confident that the dataset can be successfully used for numerous tourist and other kinds of applications.

In terms of improving the dataset, I would propose adding better categorization. For example, amenities like `restaurant`, `fast_food`, `cafe` etc. don't have common tags, and so it would be quite difficult to aggregate, analyze and provide information of all food places. Therefore, it would be very useful to have a common tag such as `amenity.category = food`.

I can see several problems that might occur during the implementation of this improvement. First of all, the dataset is quite large, which means that the effort is very high. It would be unfeasible to add the categories manually, but it is certainly possible to do this programmatically. However, programmatic approach might introduce errors to the dataset, which is why thorough validation of the categorization is crucial. Secondly, some amenities might have several categories at the same time. For example, most hotels also have restaurants, but we might not have data about it. Therefore, a lot of categories might be incomplete. It would be necessary to evaluate practical value of such categorization depending on scale of such incompleteness.

Final code

See final.py (final.py) .