OSM Data Wrangling in Seattle

September 14, 2017

1 Data Wrangling with Open Street Map(OSM)

1.1 1. Choose a city

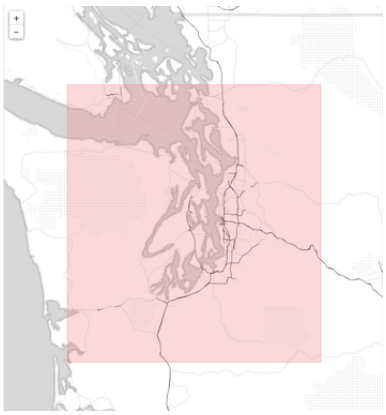
- OSM includes lots of cities spread globally.
- I could choose cities in Asia, but lots of information in written in their local languages.
- I am going to examine **Seattle WA, USA** since it is suitable to begin to wrangle with OSM at first.
- After getting done with this one, I will go for another cities like Seoul in S.Korea or Tokyo in Japan.

1.2 1-1. Map Information

- I am going to download map data (.osm) from MapZen. This website provides already prepared data file for popular cities.
- (https://goo.gl/kXjffY for Seattle WA, USA)
- When downloading OSM file, it is initially compressed.
- I need to uncompress the file first

1.3 2. Extract sample data from original

- Because uncompressed file is too large (> 1GB), it is hard to test code with it.



map_region_seattle

- In order to test functions to be in my code, I need to make sample data file extracted from the original
- The function below will do the trick.

```
In [2]: import os
        import xml.etree.ElementTree as ET

In [3]: OSM_FILE = "seattle_washington.osm"

        # Sample generation related
        SAMPLE_FILE = "seattle_washington_sample_500.osm"

In [4]: # Parameter: take every k-th top level element
        k = 500

In [5]: def get_element(osm_file, tags=('node', 'way', 'relation')):
            context = iter(ET.iterparse(osm_file, events=('start', 'end')))
            _, root = next(context)
            for event, elem in context:
                if event == 'end' and elem.tag in tags:
                    yield elem
                    root.clear()

        def generate_sample():
          with open(SAMPLE_FILE, 'wb') as output:
              output.write('<?xml version="1.0" encoding="UTF-8"?>\n')
              output.write('<osm>\n  ')

              # Write every kth top level element
              for i, element in enumerate(get_element(OSM_FILE)):
                  if i % k == 0:
                      output.write(ET.tostring(element, encoding='utf-8'))

              output.write('</osm>')

In [6]: # generate sample file
        if not os.path.exists(SAMPLE_FILE):
            generate_sample()
```

When running the code above, I would get sample data named, "seattle_washington_sample_xxx.osm"

1.4 3. OSM's XML Structure

- You can find full description about OSM's XML here (http://wiki.openstreetmap.org/wiki/OSM_XML)
- OSM basically consists of four kinds of element, node, way, tag, and nd.
- Node describes a thing

- Way describes a connection with nodes
- Tag gives additional information for node and way elements
- Nd is part of way element referencing the node by its id.
- Sample structure is shown below

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.0.2">
<bounds minlat="54.0889580" minlon="12.2487570" maxlat="54.0913900" maxlon="12
 <node id="298884269" lat="54.0901746" lon="12.2482632" user="SvenHRO" uid="468
 <node id="261728686" lat="54.0906309" lon="12.2441924" user="PikoWinter" uid="
 <node id="1831881213" version="1" changeset="12370172" lat="54.0900666" lon="1
     <tag k="name" v="Neu Broderstorf"/>
     <tag k="traffic_sign" v="city_limit"/>
 </node>
 ...
 <node id="298884272" lat="54.0901447" lon="12.2516513" user="SvenHRO" uid="468
 <way id="26659127" user="Masch" uid="55988" visible="true" version="5" changes
      <nd ref="292403538"/>
      <nd ref="298884289"/>
      ...
      <nd ref="261728686"/>
      <tag k="highway" v="unclassified"/>
      <tag k="name" v="Pastower Straße"/>
  </way>
</osm>
```

1.5 4. Audit elements

1.6 4-1. Auditing node and way elements

- node element includes attributes...
- id, lat, lon, user, uid, visible, version, changeset, timestamp
- way element includes attributes...
- id, user, uid, visible, version, changeset, timestamp
- their attributes contain not much of human editable information
- id, user, uid, visible, version, changeset, timestamp are all machine generated data
- I am not going to audit these elements for now.
- However, if I find any, I will revise this post later.

1.7 4-2. Auditing tag element

- tag element is for giving addtional information to node and way elements.
- this is mostly where user's contribution comes in, so there could be some mistakes or inconsistency.
- since there are too many tags available, I will choose some of them to audit for now.
- (you can find the entire tag set here: http://wiki.openstreetmap.org/wiki/Map_Features)
- I think the best way to audit tag element is
- list all possible values from current data
- correct inconsistencies as much as possible for now
- then if I encounter other unknown issues while running a program, go back to the first step

1.8 4-2-1. tag element where k=[maxspeed | minspeed]

- since mph is the standard speed unit in the US, values not specified with it has to be fixed.
- first, I am going to look up what values there are

```
In [7]: def audit_speed(filename):
            speed_types = []

            for event, elem in ET.iterparse(filename, events=("start",)):
                if elem.tag == "node" or elem.tag == "way":
                    for tag in elem.iter("tag"):
                        key = tag.attrib['k']
                        if key == 'maxspeed' or key == 'minspeed':
                            speed_types.append(tag.attrib['v'])

            return speed_types
```

```
In [8]: print audit_speed(SAMPLE_FILE)
```

```
['40 mph', '25 mph', '25 mph', '25 mph', '25 mph', '35 mph', '25 mph', '25 mph', '2
```

- As you can see, there are some values missing the speed unit.
- I think when the data grows larger like combining data from other countries, it is pretty important to specify what units are used to measure something.
- I am going to make a helper function to add 'mph' to those missing values.

```
In [9]: def update_speed_unit(value):
            if value.find('mph') > -1:
                return value
            else:
                value = '{} mph'.format(value)
                return value
```

```
In [10]: print update_speed_unit('20 mph')
         print update_speed_unit('20')
```

```
20 mph
20 mph
```

1.9 4-2-2. tag element where k=[phone]

- phone numbers can be written in various different ways.
- some people include national code, or others include parenthesis surrounding local code.
- it is better if all values are stored in the same form.

```
In [11]: import re
         from collections import defaultdict
```

- First, I am going to remove all special characters including '+', '-', '(', and ')' since some people use them, but some others don't
- Then, I will investigate list of lengths of phone number.
- For normal phone numbers, the length should be 10 or 11. 11 is when national code is included.

```
In [12]: def audit_phone(filename):
             phone_len = defaultdict(list)

             for event, elem in ET.iterparse(filename, events=("start",)):
                 if elem.tag == "node" or elem.tag == "way":
                     for tag in elem.iter("tag"):
                         key = tag.attrib['k']
                         if key == 'phone':
                             phone_num = re.sub(r'[\+\(\)\-\s]', '', tag.attrib['v'
                             phone_len[len(phone_num)].append(tag.attrib['v'])

             return phone_len

In [13]: phone_audit_data = audit_phone(SAMPLE_FILE)

In [14]: phone_audit_data

Out[14]: defaultdict(list,
                     {4: ['+1-253-'],
                      10: ['206-220-4240', '206-524-7951', '(425) 917-1417'],
                      11: ['+1 206-633-3411',
                       '+1-206-547-1961',
                       '+1 206 448-8677',
                       '+1-425-497-8868',
                       '+1 206-467-9200',
                       '+1 206 659-4043']})
```

- As you can see, there are 3 different lengths of phone numbers.
- The length 4 is abnormal, it doesn't mean anything. I think I should get rid of it.
- As expected, the length 10 doesn't include the national code, but the length 11 does.
- I am going to define a function to update those inconsistently written phone number to uniform shape.

- it is going to be '+NationalCode LocalCode-3Digits-4Digits'
- For abnormal cases, I will just return None so I can ignore later.

```
In [106]: def update_phone(phone_num):
              phone_num = re.sub(r'[\+\(\)\-\s]', '', phone_num)

              if len(phone_num) != 10 and \
                 len(phone_num) != 11:
                      return None

              if len(phone_num) == 10:
                   phone_num = '1{}'.format(phone_num)

              phone_num_parts = []
              phone_num_parts.append('+')
              phone_num_parts.append(phone_num[:1])
              phone_num_parts.append(' ')
              phone_num_parts.append(phone_num[1:4])
              phone_num_parts.append('-')
              phone_num_parts.append(phone_num[4:7])
              phone_num_parts.append('-')
              phone_num_parts.append(phone_num[7:])

              return ''.join(phone_num_parts)
```

```
In [107]: updated_phone_nums = []
          for length, phone_nums in phone_audit_data.items():
              for phone_num in phone_nums:
                  update_phone_num = update_phone(phone_num)

                  if update_phone_num is not None:
                      updated_phone_nums.append(update_phone_num)

          print updated_phone_nums
```

```
['+1 206-220-4240', '+1 206-524-7951', '+1 425-917-1417', '+1 206-633-3411', '+1 20
```

1.10 4-2-3. tag element where k=[addr:street]

- We use many different street names.
- We even abbreviate those names, which makes hard to read sometimes.
- I am going to look up what kind of street names are used, and what names drive the whole data to be inconsistent.

```
In [108]: STREET_TYPES_RE = re.compile(r'\b\S+\.?$', re.IGNORECASE)
```

```
In [109]: def audit_street_name(filename):
              street_types = defaultdict(set)
```

```
            for event, elem in ET.iterparse(filename, events=("start",)):
                if elem.tag == "node" or elem.tag == "way":
                    for tag in elem.iter("tag"):
                        key = tag.attrib['k']
                        if key == "addr:street":
                            street_name = tag.attrib['v']
                            match = STREET_TYPES_RE.search(street_name)
                            if match:
                                street_type = match.group()
                                street_types[street_type].add(street_name)

            return street_types
```

```
In [110]: audit_street_name(SAMPLE_FILE).keys()
```

```
Out[110]: ['Northeast',
           'Court',
           'South',
           'West',
           'Boulevard',
           'Northwest',
           'Way',
           'East',
           'Highway',
           'Southwest',
           'North',
           'Southeast',
           'Road',
           'Spur',
           'NW',
           'Loop',
           'Lane',
           'N.',
           'Drive',
           'Place',
           '104',
           'Point',
           'WY',
           'SW',
           'Street',
           'Crescent',
           'Avenue']
```

- Those values listed above are street name used in the last part of the full street names.
- Ok. Now I am going to look through what full street names are for those .

```
In [111]: #audit_street_name(SAMPLE_FILE)
```

- Here are some of the abbreviations used in the sample data and their mapped full name.
- NW: Northwest
- N.: North
- WY: Way
- SW: Southwest
- Here are some of the commonly used abbreviations
- NE: Northeast
- SE: Southeast
- S.: South
- St/St.: Street
- Rd/Rd.: Road
- Ave: Avenue
- In order to update those abbreviated street names to the full name, I need to create mapping table.

```
In [112]: STREET_TYPE_MAPPINGS = { "St"    :  "Street",
                                   "St."   :  "Street",
                                   "Rd"    :  "Road",
                                   "Rd."   :  "Road",
                                   "Ave"   :  "Avenue",
                                   "SW"    :  "Southwest",
                                   "NW"    :  "Northwest",
                                   "SE"    :  "Southeast",
                                   "NE"    :  "Northeast",
                                   "S."    :  "South",
                                   "N."    :  "North",
                                   "WY"    :  "Way"}
```

- I am going to write a function to update street name now.

```
In [113]: STREET_TYPES_RE = re.compile(r'\b\S+\.?$', re.IGNORECASE)
```

```
In [114]: def update_street_name(name):
              m = STREET_TYPES_RE.search(name)

              if m:
                  street_type = m.group()

                  try:
                      name = re.sub(street_type, STREET_TYPE_MAPPINGS[street_type],
                      return name
                  except KeyError as e:
                      return name
```

```
In [115]: updated_street_name = []

          for street_name in audit_street_name(SAMPLE_FILE).keys():
              updated_street_name.append(update_street_name(street_name))

          updated_street_name
```

```
Out[115]: ['Northeast',
           'Court',
           'South',
           'West',
           'Boulevard',
           'Northwest',
           'Way',
           'East',
           'Highway',
           'Southwest',
           'North',
           'Southeast',
           'Road',
           'Spur',
           'Northwest',
           'Loop',
           'Lane',
           'North',
           'Drive',
           'Place',
           '104',
           'Point',
           'Way',
           'Southwest',
           'Street',
           'Crescent',
           'Avenue']
```

1.11 5. Reorganize OSM into CSV (prepare for SQL)

- in order to export OSM as CSV for SQL imgration later, I need to store each tag's information in separate csv file.
- especially, tag element has to be tracked which node or way element it belongs to.
- also, nd element has to be tracked as well about which way element it belongs to.

```
In [116]: import csv
          import schema
          import codecs
          import cerberus

          SCHEMA = schema.Schema
```

```
NODES_PATH = "nodes.csv"
NODE_TAGS_PATH = "nodes_tags.csv"
WAYS_PATH = "ways.csv"
WAY_NODES_PATH = "ways_nodes.csv"
WAY_TAGS_PATH = "ways_tags.csv"

# this regex finds a string that contains problematic characters
PROBLEMCHARS = re.compile(r'[=\+\/\&\<\>\;\'\"\?\%\#\$\@\,\. \t\r\n]')
LOWER_COLON = re.compile(r'^([a-z|_]+)+:([a-z|_]+)+')

NODE_FIELDS = ['id', 'lat', 'lon', 'user', 'uid', 'version', 'changeset',
NODE_TAGS_FIELDS = ['id', 'key', 'value', 'type']
WAY_FIELDS = ['id', 'user', 'uid', 'version', 'changeset', 'timestamp']
WAY_TAGS_FIELDS = ['id', 'key', 'value', 'type']
WAY_NODES_FIELDS = ['id', 'node_id', 'position']
```

- helper functions are defined for code readibility and reproducability.
- shape_tag_element function
- this is where update for inconsistent tag's value occurs
- if key contains ":", string before ":" is the type of the tag, and string after ":" becomes the key.
- if there is no ":" in key, tag type becomes 'regular'
- this function returns a dictionary which contains..
  - key, type(tag type), id(reference which the tag belongs to node / way element)

```
In [117]: def shape_tag_element(tag_element, ref_id, default_tag_type):
              tag_attribs = {}

              key = tag_element.attrib['k']
              value = tag_element.attrib['v']

              if re.search(PROBLEMCHARS, key):
                  return None

              key_match = re.search(LOWER_COLON, key)
              if key_match:
                  key_type = key_match.group(1)
                  key_index = (key.index(key_type) + len(key_type))+1

                  tag_attribs['key'] = key[key_index: ]
                  tag_attribs['type'] = key_type
              else:
                  tag_attribs['key'] = key
                  tag_attribs['type'] = default_tag_type

              tag_attribs['id'] = ref_id
```

```
            '''
            this is where tag value update will happen
            '''
            if tag_attribs['key'] == 'maxspeed' or tag_attribs['key'] == 'minspee
                value = update_speed_unit(value)
            elif tag_attribs['key'] == 'phone':
                value = update_phone(value)
            elif tag_attribs['key'] == 'street':
                value = update_street_name(value)

            if value == None:
                return None

            tag_attribs['value'] = value

            return tag_attribs
```

- shape_tag_elements function
- this function simply iterate through all tags belonging to a node or way element and make a list of them.

```
In [118]: def shape_tag_elements(tags, parent_element, default_tag_type):
            for tag_element in parent_element.iter('tag'):
                tag_attribs = shape_tag_element(tag_element, \
                                                parent_element.attrib['id'], \
                                                default_tag_type)

                if tag_attribs != None:
                    tags.append(tag_attribs)
```

- shape_common_for_node_and_way function

```
In [119]: def shape_common_for_node_and_way(common_attribs, element):
            common_attribs['id'] = int(element.attrib['id'])
            common_attribs['uid'] = int(element.attrib['uid'])
            common_attribs['changeset'] = int(element.attrib['changeset'])
            common_attribs['user'] = element.attrib['user']
            common_attribs['version'] = element.attrib['version']
            common_attribs['timestamp'] = element.attrib['timestamp']
```

- shape_element function
- this function identifies node and way elements.

```
In [120]: def shape_nd_element(nd_element, ref_id, position):
            nd_attribs = {}

            nd_attribs['id'] = ref_id
            nd_attribs['node_id'] = nd_element.attrib['ref']
            nd_attribs['position'] = position
```

```
            return nd_attribs

In [121]: def shape_element(element, key_error_count, problem_chars=PROBLEMCHARS, d
            node_attribs = {}
            way_attribs = {}
            way_nodes = []
            tags = []

            if element.tag == 'node':
                try:
                    shape_common_for_node_and_way(node_attribs, element)
                    node_attribs['lat'] = float(element.attrib['lat'])
                    node_attribs['lon'] = float(element.attrib['lon'])

                    shape_tag_elements(tags, element, default_tag_type)
                    return {'node': node_attribs, 'node_tags': tags}
                except KeyError as e:
                    key_error_count += 1

            elif element.tag == 'way':
                try:
                    shape_common_for_node_and_way(way_attribs, element)
                    shape_tag_elements(tags, element, default_tag_type)

                    nd_position = 0
                    for nd_element in element.iter('nd'):
                        nd_attribs = shape_nd_element(nd_element, \
                                                      int(element.attrib['id']),
                                                      nd_position)
                        way_nodes.append(nd_attribs)
                        nd_position += 1

                    return {'way': way_attribs, 'way_nodes': way_nodes, 'way_tags
                except KeyError as e:
                    key_error_count += 1

In [122]: class UnicodeDictWriter(csv.DictWriter, object):
            """Extend csv.DictWriter to handle Unicode input"""

            def writerow(self, row):
                super(UnicodeDictWriter, self).writerow({
                    k: (v.encode('utf-8') if isinstance(v, unicode) else v) for k
                })

            def writerows(self, rows):
                for row in rows:
                    self.writerow(row)
```

```
In [123]: def get_element(osm_file, tags=('node', 'way', 'relation')):
              """Yield element if it is the right type of tag"""

              context = ET.iterparse(osm_file, events=('start', 'end'))
              _, root = next(context)
              for event, elem in context:
                  if event == 'end' and elem.tag in tags:
                      yield elem
                      root.clear()
```

Storing each elements' data into separate file

```
In [124]: def process_map(file_in):
              with codecs.open(NODES_PATH, 'w') as nodes_file, \
                   codecs.open(NODE_TAGS_PATH, 'w') as nodes_tags_file, \
                   codecs.open(WAYS_PATH, 'w') as ways_file, \
                   codecs.open(WAY_NODES_PATH, 'w') as way_nodes_file, \
                   codecs.open(WAY_TAGS_PATH, 'w') as way_tags_file:

                  nodes_writer = UnicodeDictWriter(nodes_file, NODE_FIELDS)
                  node_tags_writer = UnicodeDictWriter(nodes_tags_file, NODE_TAGS_F
                  ways_writer = UnicodeDictWriter(ways_file, WAY_FIELDS)
                  way_nodes_writer = UnicodeDictWriter(way_nodes_file, WAY_NODES_FI
                  way_tags_writer = UnicodeDictWriter(way_tags_file, WAY_TAGS_FIELD

                  '''
                  nodes_writer.writeheader()
                  node_tags_writer.writeheader()
                  ways_writer.writeheader()
                  way_nodes_writer.writeheader()
                  way_tags_writer.writeheader()
                  '''

                  validator = cerberus.Validator()
                  key_error_count = 0

                  for element in get_element(file_in, tags=('node', 'way')):
                      el = shape_element(element, key_error_count)
                      if el:
                          if element.tag == 'node':
                              nodes_writer.writerow(el['node'])
                              node_tags_writer.writerows(el['node_tags'])
                          elif element.tag == 'way':
                              ways_writer.writerow(el['way'])
                              way_nodes_writer.writerows(el['way_nodes'])
                              way_tags_writer.writerows(el['way_tags'])

                  print 'number of node or way element that encountered KeyError {}
```

```
In [125]: process_map(OSM_FILE)
```

```
number of node or way element that encountered KeyError 0
```

1.12 6. Importing CSV into SQLite

Database Schema for SQLite3

```
CREATE TABLE nodes (
    id INTEGER PRIMARY KEY NOT NULL,
    lat REAL,
    lon REAL,
    user TEXT,
    uid INTEGER,
    version INTEGER,
    changeset INTEGER,
    timestamp TEXT
);

CREATE TABLE nodes_tags (
    id INTEGER,
    key TEXT,
    value TEXT,
    type TEXT,
    FOREIGN KEY (id) REFERENCES nodes(id)
);

CREATE TABLE ways (
    id INTEGER PRIMARY KEY NOT NULL,
    user TEXT,
    uid INTEGER,
    version TEXT,
    changeset INTEGER,
    timestamp TEXT
);

CREATE TABLE ways_tags (
    id INTEGER NOT NULL,
    key TEXT NOT NULL,
    value TEXT NOT NULL,
    type TEXT,
    FOREIGN KEY (id) REFERENCES ways(id)
);

CREATE TABLE ways_nodes (
    id INTEGER NOT NULL,
    node_id INTEGER NOT NULL,
```

```
    position INTEGER NOT NULL,
    FOREIGN KEY (id) REFERENCES ways(id),
    FOREIGN KEY (node_id) REFERENCES nodes(id)
);
```

importing data from csv into the tables

```
sqlite> .tables nodes       nodes_tags  ways        ways_nodes
ways_tags sqlite> .mode csv sqlite> .import ./nodes.csv nodes sqlite>
.import ./nodes_tags.csv nodes_tags sqlite> .import ./ways.csv
ways sqlite> .import ./ways_nodes.csv ways_nodes sqlite> .import
./ways_tags.csv ways_tags
```

checking first few lines for each table

```
sqlite> SELECT * FROM nodes LIMIT 2;
25832758|48.4364741|-123.3129318|CoreyBurger|6009  |4|9689672|2011-10-29T23:34:53Z
25839536|48.435199 |-123.3288383|alester    |307202|3|9568618|2011-10-16T00:43:37Z

sqlite> SELECT * FROM nodes_tags LIMIT 2;
27170286|created_by|YahooApplet 1.0|regular
30139083|highway   |traffic_signals|regular

sqlite> SELECT * FROM ways LIMIT 2;
4388496|alester|307202 |2|8212025 |2011-05-22T02:31:47Z
4736718|Madrona|3305376|4|34783992|2015-10-21T17:39:39Z

sqlite> SELECT * FROM ways_nodes LIMIT 2;
4388496|1295593467|0
4388496|26793412  |1

sqlite> SELECT * FROM ways_tags LIMIT 2;
4388496|name   |Arden Road |regular
4388496|highway|residential|regular
```

```
sqlite> .save wrangle_us.db sqlite> .mode column sqlite> .headers on
```

1.13 7. Perform some queries

1.14 7-1. Number of contributors to node and way elements

```
sqlite>
SELECT COUNT(DISTINCT user) as num_of_user
FROM
    (
     SELECT user
     FROM nodes
     UNION ALL
         SELECT user
         FROM ways
    )
```

```
;

num_of_user
-----------
631
```

As expected lots of user (631) did participated in to collect and input data to build a map data for this huge city

1.15 7-2. Top 10 contributors

```
sqlite>
SELECT DISTINCT user, COUNT(*) as num_of_contrib
FROM
    (
     SELECT user
     FROM nodes
     UNION ALL
        SELECT user
        FROM ways
    )
GROUP BY user
ORDER BY num_of_contrib DESC
LIMIT 10;

user         num_of_contrib
-----------  --------------
Glassman     2602
SeattleImp   1472
tylerritch   1316
woodpeck_f   1149
alester      713
Omnific      628
Glassman_I   453
CarniLvr79   415
STBrenden    415
Brad Meteo   373
```

It looks like Glassman did make huge contribution.

1.16 7-3. The most included tag type under node and way respectively

```
sqlite>
SELECT type as type_in_node, COUNT(*) as num
FROM nodes_tags
GROUP BY type
ORDER BY num DESC
LIMIT 2;
```

```
type_in_node  num
------------  ----------
regular       1191
addr          978
```

The "regular" is the most included tag type. It is not surprising since regular tag is something which doesn't provide additional information included in the key. There are not many tags available which come with additional information. The addr tag is the second most popular one. It looks reseonable to have lots of address information to describe a map.

```
sqlite>
SELECT type as type_in_way, COUNT(*) as num
FROM ways_tags
GROUP BY type
ORDER BY num DESC
LIMIT 2;
```

```
type_in_way   num
------------  ----------
regular       3518
tiger         1560
```

The tag type, "tiger" is the most included tag type after regular type. I have searched what this type of tag is, and I found the tiger tag type means *"The string 'tiger' is the prefix of keys that are part of the TIGER import of the United States."*. **It looks like a sort of imported tag information from other map's dataset. This raised another question if this tag type is used to form the node elements as well.**

1.17 7-4. tiger type tag looks popular in way element, how about in node element?

```
sqlite>
SELECT type as type_in_node, COUNT(*) as num
FROM nodes_tags
GROUP BY type
ORDER BY num DESC
```

```
type_in_node  num
------------  ----------
regular       1191
addr          978
gtfs          26
gnis          23
sdot          20
source        17
is_in         3
name          3
seamark       3
checked_exis  2
```

```
disused       1
toilets       1
```

This looked strange at first because the tiger tag type is never included in node elements, so I looked at the documentation. This is what I found. *"In 2010 tiger tags were removed from about 177 million nodes, affected tags were: tiger:county=, *tiger:tlid*= and tiger:upload_uuid=*. See TIGER fixup/node tags for details. As a result, tiger keys are now almost exclusively found on ways."* **now it makes sense why tiger tag doesn't appear at all.

1.18 7-5. Top 5 tag keys whose type is "regular" under node and way respectively, and in combination of the two.

```
sqlite>
SELECT key as key_in_node, COUNT(*) as num
FROM nodes_tags
WHERE type='regular'
GROUP BY key
ORDER BY num DESC
LIMIT 5;

key_in_node  num
-----------  ----------
source       420
created_by   137
highway      100
power        90
name         68
```

Source, created_by, and name don't give much information to understand. However, the count of highway tag explains that lots of nodes are built around highway. Also, lots of power tags are included to descrige node elements. Let me look more detail what values for power tags are used.

```
SELECT value as value_in_power, COUNT(*) as num
FROM nodes_tags
WHERE type='regular'
AND key='power'
ORDER BY num DESC;

value_in_power  num
--------------  ----------
pole            90
```

Surprisingly, the only one value, "pole", is used under "power" tag key. This is what pole is according to the documentation. *Poles supporting low to medium voltage lines (power=minor_line) and high voltage lines (power=line) up to 161,000 volts (161 kV).* **I don't quiet understand why the only pole value is included, it makes me to think this information is incomplete**

```
sqlite>
SELECT key as key_in_way, COUNT(*) as num
FROM ways_tags
WHERE type='regular'
GROUP BY key
ORDER BY num DESC
LIMIT 5;

key_in_way  num
----------  ----------
building    683
highway     615
source      561
name        399
service     107
```

Building, highway, and service are informative data to describe ways.

```
sqlite>
SELECT key, COUNT(*) as num
FROM
    (
     SELECT key
     FROM nodes_tags
     UNION ALL
        SELECT key
        FROM ways_tags
        WHERE type='regular'
    )
GROUP BY key
ORDER BY num DESC
LIMIT 5;

key         num
----------  ----------
source      981
highway     715
building    685
name        467
housenumbe  260
```

For both way and node elements, highway, building, and housenumber seems like to be used most frequently. It looks a bit strange the number of housenumber is a way smaller than the number of buildings though. I found that "addr:housenumber" tag also describes housenumber information, so it might be possible to have same informational data in two different tags.

1.19 8. Conclusion

While doing this project, I have learned the followings. - How to investigate XML based dataset - How to parse XML elements with their attributes - How to make inconsistent data in the XML to be consistent - How to export re-organized data into CSV format. - How to export CSVs into SQLite based RDB. - How to run SQL syntax queries to investigate the dataset more expressively and efficiently.

The OSM data gives a lot of information, but the information is formed by human being not machine generated. It means it contains lots of inconsistent and not well-organized data format sometimes. In order to find more meaningful information out of the dataset, it has to be cleaned so that data analysts can trust it. I think I became more confident to achieve converting inconsistent data to consistent data through this project.

In [ ]: